JFP **29**, e12, 24 pages, 2019.

doi:10.1017/S095679681900008X

Perturbation confusion in forward automatic differentiation of higher-order functions

OLEKSANDR MANZYUK[1], BARAK A. PEARLMUTTER, ALEXEY ANDREYEVICH RADUL[2] and DAVID R. RUSH[3]

Department of Computer Science and Hamilton Institute, Maynooth University, Co. Kildare, Ireland
(*e-mails:* manzyuk@gmail.com, barak@pearlmutter.net, axch@alum.mit.edu, kumoyuki@gmail.com)

JEFFREY MARK SISKIND

School of Electrical and Computer Engineering, Purdue University, West Lafayette, IN 47907-2035, USA
(*e-mail:* qobi@purdue.edu)

Abstract

Automatic differentiation (AD) is a technique for augmenting computer programs to compute derivatives. The essence of AD in its forward accumulation mode is to attach perturbations to each number, and propagate these through the computation by overloading the arithmetic operators. When derivatives are nested, the distinct derivative calculations, and their associated perturbations, must be distinguished. This is typically accomplished by creating a unique tag for each derivative calculation and tagging the perturbations. We exhibit a subtle bug, present in fielded implementations which support derivatives of higher-order functions, in which perturbations are confused *despite* the tagging machinery, leading to incorrect results. The essence of the bug is as follows: a unique tag is needed for each derivative calculation, but in existing implementations unique tags are created when taking the derivative of a function at a point. When taking derivatives of higher-order functions, these need not correspond! We exhibit a simple example: a higher-order function f whose derivative at a point x, namely $f'(x)$, is itself a function which calculates a derivative. This situation arises naturally when taking derivatives of curried functions. Two potential solutions are presented, and their deficiencies discussed. One uses eta expansion to delay the creation of fresh tags in order to put them into one-to-one correspondence with derivative calculations. The other wraps outputs of derivative operators with tag substitution machinery. Both solutions seem very difficult to implement without violating the desirable complexity guarantees of forward AD.

1 Introduction

The classical univariate derivative of a function $f : \mathbb{R} \to \mathbb{R}$ is a function $f' : \mathbb{R} \to \mathbb{R}$ (Leibniz, 1684; Newton, 1704). Multivariate or vector calculus extends the notion of derivative to functions whose domains and/or ranges are aggregates, that is vectors, introducing notions like gradients, Jacobians, and Hessians. Differential geometry further

[1]Current affiliation: Facebook.
[2]Current affiliation: Google AI.
[3]Current address: Dunlavin, Ireland.

extends the notion of derivatives to functions whose domains and/or ranges are—or can contain—functions.

Automatic differentiation (AD) is a collection of methods for computing the derivative of a function at a point when the function is expressed as a computer program (Griewank & Walther, 2008). These techniques, once pursued mainly by a small quiet academic community, have recently moved to the forefront of deep learning, where more expressive languages can spawn new industries, efficiency improvements can save billions of dollars, and errors can have far-reaching consequences.

From its earliest days, AD has supported functions whose domains and/or ranges are aggregates. There is currently interest from application programmers (machine learning in particular) in applying AD to higher-order functions. Here, we consider extending AD to support functions whose domains and/or ranges are functions. This is natural: we wish AD to be completely general and apply in an unrestricted fashion to correctly compute the derivative of all programs that compute differentiable mathematical functions. This includes applying to functions whose domain and/or ranges include the entire space of data types supported by programming languages, including not only aggregates but also functions. In doing so, we uncover a subtle bug. Although for expository purposes we present the bug in the context of forward AD (Wengert, 1964), the underlying issue can also manifest itself with other AD modes, including reverse AD (Speelpenning, 1980) of higher-order functions. The bug is insidious: it can lead to production of incorrect results without warning. We present and discuss the relative merits of two fixes, and exhibit code implementing them.

Our solutions are not ideal. While we believe that the solutions will always produce the correct result, they can foil both the space and time complexity guarantees of forward AD described in the next section.

Let $\mathbb{D}$ denote the true mathematical derivative operator. $\mathbb{D}$ is classically defined for first-order functions $\mathbb{R} \to \mathbb{R}$ in terms of limits, and thus this classical definition does not lend itself to direct implementation.

$$\mathbb{D}\, f = f' \qquad\qquad \text{where } f'(x) = \lim_{\epsilon \to 0} \frac{f(x+\epsilon) - f(x)}{\epsilon} \tag{1}$$

We seek to materialize $\mathbb{D}$ as a program construct $\mathscr{D}$. We can view this classical limit definition as a *specification* of $\mathscr{D}$ and proceed to develop an *implementation* of $\mathscr{D}$. Below, we use $=$ to denote mathematical equality, $\triangleq$ to denote definition of program constructs, and $\Longrightarrow$ to denote evaluation.

One can extend $\mathbb{D}$ to functions $\mathbb{R} \to \alpha$, where:

$$\alpha ::= \mathbb{R} \mid \alpha_1 \to \alpha_2 \tag{2}$$

We first focus on this extension in Sections 2–8. We consider further extension to functions $\alpha_1 \to \alpha_2$ in Section 9. Since by (2) any type α must be of the form $\alpha_1 \to \cdots \to \alpha_n \to \mathbb{R}$, functions $\mathbb{R} \to \alpha$ can be viewed as multivariate functions $\mathbb{R} \to \alpha_2 \to \cdots \to \alpha_n \to \mathbb{R}$ whose first argument domain is $\mathbb{R}$ and whose range is $\mathbb{R}$. We take $\mathbb{D}\, f$ where $f : \mathbb{R} \to \alpha_2 \to \cdots \to \alpha_n \to \mathbb{R}$ to be the partial derivative with respect to the first argument.

$$\mathbb{D}\, f = \frac{\partial f(x_1, x_2, \ldots, x_n)}{\partial x_1} \tag{3}$$

We will see below that past work has implemented a $\mathscr{D}$ that appears to coincide with the specification $\mathbb{D}$ in (1) for functions $\mathbb{R} \to \mathbb{R}$, but this past implementation fails to coincide with the specification $\mathbb{D}$ in (3) for functions $\mathbb{R} \to \alpha$. We then proceed to demonstrate two new implementations of $\mathscr{D}$ that do appear to coincide.

2 Forward AD as differential algebra

Forward AD can be formulated as differential algebra (Karczmarczuk, 2001). Its essence is as follows.

The purely arithmetic theory of complex numbers as pairs of real numbers was introduced by Hamilton (1837). These form an algebra over two-term polynomials $a + b\mathrm{i}$ where $\mathrm{i}^2 = -1$. Arithmetic proceeds by simple rules, derived algebraically.

$$(a + b\mathrm{i}) + (c + d\mathrm{i}) = (a + c) + (b + d)\mathrm{i} \tag{4a}$$

$$(a + b\mathrm{i})(c + d\mathrm{i}) = ac + (ad + bc)\mathrm{i} + bd\mathrm{i}^2 = (ac - bd) + (ad + bc)\mathrm{i} \tag{4b}$$

Complex numbers can be implemented in a computer as ordered pairs (a, b), sometimes called Argand pairs. Since arithmetic over complex numbers is defined in terms of arithmetic over the reals, the above rules imply that computation over complex numbers is closed.

Clifford (1873) introduced *dual numbers* of the form $a + b\epsilon$. In a dual number, the coefficient of ϵ is called a perturbation or a *tangent*. These can similarly be viewed as an algebra over two-term polynomials where $\epsilon^2 = 0$ but $\epsilon \neq 0$. Arithmetic over dual numbers is again defined by simple rules derived algebraically.

$$(a + b\epsilon) + (c + d\epsilon) = (a + c) + (b + d)\epsilon \tag{5a}$$

$$(a + b\epsilon)(c + d\epsilon) = ac + (ad + bc)\epsilon + bd\epsilon^2 = ac + (ad + bc)\epsilon \tag{5b}$$

Again, dual numbers can be implemented in a computer as ordered pairs (a, b). Again, since arithmetic over dual numbers is defined in terms of arithmetic over the reals, the above rules imply that computation over dual numbers is closed.

The essence of forward AD is viewing dual numbers as truncated two-term power series. Since, following Taylor (1715), $f(x_0 + x_1\epsilon + O(\epsilon^2)) = f(x_0) + x_1 f'(x_0)\epsilon + O(\epsilon^2)$, applying f to a dual number $a + 1\epsilon$ will yield a dual number $f(a) + f'(a)\epsilon$. This leads to the following method for computing derivatives of functions $f : \mathbb{R} \to \mathbb{R}$ expressed as computer programs.

- Arrange for the programming language to support dual numbers and arithmetic thereupon.
- To compute f' at a point a,
  1. form $a + 1\epsilon$,
  2. apply f to $a + 1\epsilon$ to obtain a result $f(a) + f'(a)\epsilon$, and
  3. extract the tangent, $f'(a)$, from the result.

Step 2 constitutes a nonstandard interpretation of the arithmetic basis functions with (5a) and (5b). This can be implemented in various ways, for example, overloading or source-code transformation. Further, dual numbers can be represented in various ways, for example, as unboxed flattened values or as boxed values referenced through pointers. These different implementation strategies do not concern us here. While different implementation strategies have different costs, what we discuss applies to all strategies.

It is convenient to encapsulate steps 1–3 as a higher-order function $\mathscr{D} : f \mapsto f'$. Indeed, that seems to be one of the original motivations for the development of the lambda calculus (Church, 1941, ¶4). We can do this with the following code that implements $\mathscr{D}$.

$$\mathbf{tg}\ a \stackrel{\triangle}{=} 0 \qquad a : \mathbb{R} \tag{6a}$$

$$\mathbf{tg}\ (a + b\epsilon) \stackrel{\triangle}{=} b \tag{6b}$$

$$\mathscr{D}\, f\, x \stackrel{\triangle}{=} \mathbf{tg}\ (f\ (x + 1\epsilon)) \tag{6c}$$

Here, $x + 1\epsilon$ denotes step 1 above, that is, constructing a dual number, and $\mathbf{tg}\ (a + b\epsilon)$ denotes step 3 above, that is, extracting the tangent of a dual number. Equation (6a) handles the case where the output of f is independent of the input x.

Forward AD provides certain complexity guarantees. Steps 1 and 3 take unit time. Step 2 introduces no more than a constant factor increase in both the space and time complexity of executing f under a nonstandard interpretation. Thus computing $f\ x$ and $\mathscr{D}\, f\, x$ have the same space and time complexity.

3 Tagging dual numbers to avoid perturbation confusion

Siskind & Pearlmutter (2008) discuss a problem with the above. It is natural to nest application of $\mathscr{D}$. Doing so would allow taking higher-order derivatives and, more generally, derivatives of functions that take derivatives of other functions.

$$\mathscr{D}\ (\lambda x\,.\ \ldots \mathscr{D}\ (\lambda y\,.\ \ldots)\ \ldots)\ \ldots \tag{7}$$

This can lead to *perturbation confusion* (Siskind & Pearlmutter, 2005, Section 2, Eqs. (4)–(11)), yielding an incorrect result. The essence of perturbation confusion is that each invocation of $\mathscr{D}$ must perform its computation over a distinct differential algebra. While it is possible to reject programs that would exhibit perturbation confusion using static typing (Buckwalter, 2007; Kmett, 2010), and static typing can be used to yield the desired correct result in some cases with some user annotation (Shan, 2008), no static method is known that can yield the desired correct result in all cases without any annotation. It is possible, however, to get the correct result in all cases (except, as we shall see, when taking derivatives of functions whose ranges are functions) without user annotation, by redefining **tg** and $\mathscr{D}$ to *tag* dual numbers with distinct ϵs to obtain distinct differential algebras (or equivalently, distinct generators in a differential algebra) introduced by different invocations of $\mathscr{D}$ (Lavendhomme, 1996). We will indicate different tags by different subscripts on ϵ, and use ε to denote a variable that is bound to an ϵ.

$$\mathbf{tg}\ \varepsilon\, a \stackrel{\triangle}{=} 0 \qquad a : \mathbb{R} \tag{8a}$$

$$\mathbf{tg}\ \varepsilon\ (a + b\varepsilon) \stackrel{\triangle}{=} b \tag{8b}$$

$$\mathbf{tg}\ \varepsilon_1\ (a + b\varepsilon_2) \stackrel{\triangle}{=} (\mathbf{tg}\ \varepsilon_1\ a) + (\mathbf{tg}\ \varepsilon_1\ b)\varepsilon_2 \qquad \varepsilon_1 \neq \varepsilon_2 \tag{8c}$$

$$\mathscr{D}\, f\, x \stackrel{\triangle}{=} \mathbf{fresh}\ \varepsilon\ \mathbf{in}\ \mathbf{tg}\ \varepsilon\ (f\ (x + 1\varepsilon)) \tag{8d}$$

These redefine (6a)–(6c). Here, the tags are generated dynamically. Many systems employ this approach.[1] Many of these systems are implemented in “mostly functional languages,” like SCHEME, ML, F♯, PYTHON, LUA, and JULIA, and are intended to be used with pure subsets of these languages.

Prior to this change, that is with only a single ϵ, the values a and b in a dual number $a + b\epsilon$ would be real numbers. With this change, that is with multiple ϵs, the values a and b in a dual number $a + b\epsilon_1$ can be dual numbers over ϵ_2 where $\epsilon_2 \neq \epsilon_1$. Such a tree of dual numbers will contain real numbers in its leaves and will contain a given ϵ only once along each path from the root to the leaves. Equation (8c) provides the ability to extract the tangent of an ϵ that might not be at the root of the tree.

4 Extending to functions whose range is a function

If one applies $\mathscr{D}$ to a function f whose range is a function, $f\ (x + 1\varepsilon)$ in (8d) will yield a function. In this higher-order case, when f returns a function g, an invocation $\mathscr{D}\ f\ x$ yields a function $\bar{g}$ which performs a derivative calculation when invoked. It will not be possible to extract the tangent of this with **tg** as implemented by (8a)–(8c). The definition of **tg** can be augmented to handle this case by post-composition.[2]

$$\mathbf{tg}\ \varepsilon\ \bar{g} \overset{\triangle}{=} (\mathbf{tg}\ \varepsilon) \circ \bar{g} \qquad \bar{g} \text{ is a function} \tag{8e}$$

However, this extension (alone) is flawed, as we proceed to demonstrate.

5 A bug

Consider the following commonly occurring mathematical situation. We define an offset operator:

$$s : \mathbb{R} \to (\mathbb{R} \to \mathbb{R}) \to \mathbb{R} \to \mathbb{R}$$
$$s\, u\, f\, x \overset{\triangle}{=} f\ (x + u) \tag{9}$$

The derivative of s at zero should be the same as the derivative operator, that is, $\mathbb{D}\ s\ 0 = \mathbb{D}$, since:

$$(\forall f)(\forall y)\mathbb{D}\, s\, 0\, f\, y = \frac{\partial}{\partial u}[\, s\, u\, f\, y\,]_{u=0} = \frac{\partial}{\partial u}[f(\,y + u)\,]_{u=0} = f'(\,y) = \mathbb{D}\, f\, y \tag{10a}$$
$$\Longleftrightarrow \qquad \{\text{eta}\}$$
$$(\forall f)\mathbb{D}\, s\, 0\, f = \mathbb{D}\, f \tag{10b}$$
$$\Longleftrightarrow \qquad \{\text{eta}\}$$
$$\mathbb{D}\, s\, 0 = \mathbb{D} \tag{10c}$$

[1] For example, SCMUTILS (Sussman *et al.*, 1997b,a), a software package that accompanies a textbook on classical mechanics (Sussman *et al.*, 2001) as well as a textbook on differential geometry (Sussman *et al.*, 2013), the systems of Farr (2006), Siskind & Pearlmutter (2005, 2008), Pearlmutter & Siskind (2007, 2008), R6RS-AD (`https://github.com/qobi/R6RS-AD`), DIFFSHARP (Baydin *et al.*, 2016), HIPS AUTOGRAD (Maclaurin *et al.*, 2015a), TORCH AUTOGRAD (`https://github.com/twitter/torch-autograd`), and JULIADIFF (`http://www.juliadiff.org/ForwardDiff.jl/stable/user/api.html`).

[2] Justification of this post-composition is given in Section 9 which describes the relevant constructs from differential geometry.

Thus, if we define

$$\hat{\mathscr{D}} \triangleq \mathscr{D}\, s\, 0 \tag{11}$$

we would hope that $\hat{\mathscr{D}} = \mathscr{D}$. However, we exhibit an example where it does not.

We can compute $\hat{\mathscr{D}}\,(\hat{\mathscr{D}}\,h)\,y$ for $h : \mathbb{R} \to \mathbb{R}$ with simple reduction steps:

$$\begin{aligned}
&\hat{\mathscr{D}} \\
\Longrightarrow\quad & \{\text{by (11)}\} \\
&\mathscr{D}\, s\, 0 && \text{(12a)} \\
\Longrightarrow\quad & \{\text{by (8d)}\} \\
&\textbf{fresh}\ \varepsilon\ \textbf{in}\ \textbf{tg}\ \varepsilon\ (s\,(0 + 1\varepsilon)) && \text{(12b)} \\
\Longrightarrow\quad & \{\text{allocate a fresh tag } \epsilon_0\text{; this is problematic; see discussion below}\} \\
&\textbf{tg}\ \epsilon_0\ (s\,(0 + 1\epsilon_0)) && \text{(12c)} \\
\Longrightarrow\quad & \{\text{by (9)}\} \\
&\textbf{tg}\ \epsilon_0\ (\lambda f\,.\,\lambda x\,.\,(f\,(x + 1\epsilon_0))) && \text{(12d)} \\
\Longrightarrow\quad & \{\text{by (8e)}\} \\
&(\textbf{tg}\ \epsilon_0) \circ (\lambda f\,.\,\lambda x\,.\,(f\,(x + 1\epsilon_0))) && \text{(12e)} \\
\Longrightarrow\quad & \{\text{postcompose}\} \\
&\lambda f\,.\,\lambda x\,.\,\textbf{tg}\ \epsilon_0\ (f\,(x + 1\epsilon_0)) && \text{(12f)}
\end{aligned}$$

$$\begin{aligned}
&\hat{\mathscr{D}}\,(\hat{\mathscr{D}}\,h)\,y \\
\Longrightarrow\quad & \{\text{substitute (12f) for } \hat{\mathscr{D}}\} \\
&(\lambda f\,.\,\lambda x\,.\,\textbf{tg}\ \epsilon_0\ (f\,(x + 1\epsilon_0)))\,((\lambda f\,.\,\lambda x\,.\,\textbf{tg}\ \epsilon_0\ (f\,(x + 1\epsilon_0)))\,h)\,y && \text{(12g)} \\
\Longrightarrow\quad & \{\text{beta reduce}\} \\
&(\lambda f\,.\,\lambda x\,.\,\textbf{tg}\ \epsilon_0\ (f\,(x + 1\epsilon_0)))\,(\lambda x\,.\,\textbf{tg}\ \epsilon_0\ (h\,(x + 1\epsilon_0)))\,y && \text{(12h)} \\
\Longrightarrow\quad & \{\text{beta reduce}\} \\
&(\lambda x\,.\,\textbf{tg}\ \epsilon_0\ ((\lambda x\,.\,\textbf{tg}\ \epsilon_0\ (h\,(x + 1\epsilon_0)))\,(x + 1\epsilon_0)))\,y && \text{(12i)} \\
\Longrightarrow\quad & \{\text{beta reduce}\} \\
&\textbf{tg}\ \epsilon_0\ ((\lambda x\,.\,\textbf{tg}\ \epsilon_0\ (h\,(x + 1\epsilon_0)))\,(y + 1\epsilon_0)) && \text{(12j)} \\
\Longrightarrow\quad & \{\text{beta reduce}\} \\
&\textbf{tg}\ \epsilon_0\ (\textbf{tg}\ \epsilon_0\ (h\,((y + 1\epsilon_0) + 1\epsilon_0))) && \text{(12k)} \\
\Longrightarrow\quad & \{\text{add dual numbers}\} \\
&\textbf{tg}\ \epsilon_0\ (\textbf{tg}\ \epsilon_0\ (h\,(y + 2\epsilon_0))) && \text{(12l)} \\
\Longrightarrow\quad & \{\text{apply } h \text{ to a dual number}\} \\
&\textbf{tg}\ \epsilon_0\ (\textbf{tg}\ \epsilon_0\ (h(y) + 2h'(y)\epsilon_0)) && \text{(12m)} \\
\Longrightarrow\quad & \{\text{by (8b)}\} \\
&\textbf{tg}\ \epsilon_0\ (2h'(y)) && \text{(12n)} \\
\Longrightarrow\quad & \{\text{by (8a)}\} \\
&0 && \text{(12o)}
\end{aligned}$$

This went wrong, yielding 0 instead of $h''(y)$.

$$\hat{\mathscr{D}}\,(\hat{\mathscr{D}}\,h)\,y \Longrightarrow 0 \neq \mathbb{D}\,(\mathbb{D}\,h)\,y = h''(y) \tag{13}$$

The process of allocating a fresh tag in step (12d) was problematic. The proper way to handle such fresh tag allocation might be to use nominal logic (Pitts, 2003), perhaps in a dependent-type-theoretic variant (Cheney, 2012). Below, we offer alternate mechanisms that are suitable for use in programming-language implementations that lack type systems that support first class names and binding.

This is not an artificial example. It is quite natural to construct an x-axis differential operator and apply it to a two-dimensional function twice, along the x and then y axis directions, by applying the operator, flipping the axes, and applying the operator again, thus creating precisely this sort of cascaded use of a defined differential operator.

6 The root cause of the bug

This incorrect result was due to the tag ϵ_0 being generated exactly *once*, in (12b), when $\hat{\mathscr{D}}$ was calculated from $\mathscr{D}\,s\,0$ as (12a)–(12f) using the definition (11). The invocation $\mathscr{D}\,s\,0$ is the point at which a fresh tag is introduced; early instantiation can result in reuse of the same tag in logically distinct derivative calculations. Here, the first derivative and the second derivative become confused at (12l). We have two nested applications of **tg** for ϵ_0, but for correctness these should be distinctly tagged: ϵ_0 versus ϵ_1.

This can be accomplished by making two copies of $\hat{\mathscr{D}}$ by evaluating $\mathscr{D}\,s\,0$ twice. Performing an analogous computation with two copies of $\hat{\mathscr{D}}$ yields the correct result.

$$\begin{aligned}
&\hat{\mathscr{D}}_0\\
\Longrightarrow&\qquad\{\text{repeat (12a)}\}\\
&\mathscr{D}\,s\,0 &&\text{(14a)}\\
\Longrightarrow&\qquad\{\text{repeat (12b)}\}\\
&\textbf{fresh}\;\varepsilon\;\textbf{in}\;\;\textbf{tg}\;\varepsilon\,(s\,(0+1\varepsilon)) &&\text{(14b)}\\
\Longrightarrow&\qquad\{\text{repeat (12c)}\}\\
&\textbf{tg}\;\epsilon_0\;(s\,(0+1\epsilon_0)) &&\text{(14c)}\\
\Longrightarrow&\qquad\{\text{repeat (12d)}\}\\
&\textbf{tg}\;\epsilon_0\;(\lambda f\,.\,\lambda x\,.\,(f\,(x+1\epsilon_0))) &&\text{(14d)}\\
\Longrightarrow&\qquad\{\text{repeat (12e)}\}\\
&(\textbf{tg}\;\epsilon_0)\circ(\lambda f\,.\,\lambda x\,.\,(f\,(x+1\epsilon_0))) &&\text{(14e)}\\
\Longrightarrow&\qquad\{\text{repeat (12f)}\}\\
&\lambda f\,.\,\lambda x\,.\;\textbf{tg}\;\epsilon_0\;(f\,(x+1\epsilon_0)) &&\text{(14f)}
\end{aligned}$$

$$\begin{aligned}
&\hat{\mathscr{D}}_1\\
\Longrightarrow&\qquad\{\text{repeat (12a)}\}\\
&\mathscr{D}\,s\,0 &&\text{(14g)}\\
\Longrightarrow&\qquad\{\text{repeat (12b)}\}\\
&\textbf{fresh}\;\varepsilon\;\textbf{in}\;\;\textbf{tg}\;\varepsilon\,(s\,(0+1\varepsilon)) &&\text{(14h)}
\end{aligned}$$

$$\Longrightarrow \quad \{\text{repeat (12c)}\}$$
$$\mathbf{tg}\ \epsilon_1\ (s\,(0+1\epsilon_1)) \tag{14i}$$
$$\Longrightarrow \quad \{\text{repeat (12d)}\}$$
$$\mathbf{tg}\ \epsilon_1\ (\lambda f\,.\,\lambda x\,.\,(f\,(x+1\epsilon_1))) \tag{14j}$$
$$\Longrightarrow \quad \{\text{repeat (12e)}\}$$
$$(\mathbf{tg}\ \epsilon_1)\circ(\lambda f\,.\,\lambda x\,.\,(f\,(x+1\epsilon_1))) \tag{14k}$$
$$\Longrightarrow \quad \{\text{repeat (12f)}\}$$
$$\lambda f\,.\,\lambda x\,.\ \mathbf{tg}\ \epsilon_1\ (f\,(x+1\epsilon_1)) \tag{14l}$$

$$\hat{\mathscr{D}}_0\,(\hat{\mathscr{D}}_1\,h)\,y$$
$$\Longrightarrow \quad \{\text{substitute (14f) and (14l) for } \hat{\mathscr{D}}\}$$
$$(\lambda f\,.\,\lambda x\,.\ \mathbf{tg}\ \epsilon_0\ (f\,(x+1\epsilon_0)))\,((\lambda f\,.\,\lambda x\,.\ \mathbf{tg}\ \epsilon_1\ (f\,(x+1\epsilon_1)))\,h)\,y \tag{14m}$$
$$\Longrightarrow \quad \{\text{beta reduce}\}$$
$$(\lambda f\,.\,\lambda x\,.\ \mathbf{tg}\ \epsilon_0\ (f\,(x+1\epsilon_0)))\,(\lambda x\,.\ \mathbf{tg}\ \epsilon_1\ (h\,(x+1\epsilon_1)))\,y \tag{14n}$$
$$\Longrightarrow \quad \{\text{beta reduce}\}$$
$$(\lambda x\,.\ \mathbf{tg}\ \epsilon_0\ ((\lambda x\,.\ \mathbf{tg}\ \epsilon_1\ (h\,(x+1\epsilon_1)))\,(x+1\epsilon_0)))\,y \tag{14o}$$
$$\Longrightarrow \quad \{\text{beta reduce}\}$$
$$\mathbf{tg}\ \epsilon_0\ ((\lambda x\,.\ \mathbf{tg}\ \epsilon_1\ (h\,(x+1\epsilon_1)))\,(y+1\epsilon_0)) \tag{14p}$$
$$\Longrightarrow \quad \{\text{beta reduce}\}$$
$$\mathbf{tg}\ \epsilon_0\ (\mathbf{tg}\ \epsilon_1\ (h\,((y+1\epsilon_0)+1\epsilon_1))) \tag{14q}$$
$$\Longrightarrow \quad \{\text{apply } h \text{ to a dual number}\}$$
$$\mathbf{tg}\ \epsilon_0\ (\mathbf{tg}\ \epsilon_1\ (h(y+1\epsilon_0)+h'(y+1\epsilon_0)\epsilon_1)) \tag{14r}$$
$$\Longrightarrow \quad \{\text{apply } h \text{ to a dual number}\}$$
$$\mathbf{tg}\ \epsilon_0\ (\mathbf{tg}\ \epsilon_1\ ((h(y)+h'(y)\epsilon_0)+h'(y+1\epsilon_0)\epsilon_1)) \tag{14s}$$
$$\Longrightarrow \quad \{\text{apply } h \text{ to a dual number}\}$$
$$\mathbf{tg}\ \epsilon_0\ (\mathbf{tg}\ \epsilon_1\ ((h(y)+h'(y)\epsilon_0)+(h'(y)+h''(y)\epsilon_0)\epsilon_1)) \tag{14t}$$
$$\Longrightarrow \quad \{\text{by (8b)}\}$$
$$\mathbf{tg}\ \epsilon_0\ (h'(y)+h''(y)\epsilon_0) \tag{14u}$$
$$\Longrightarrow \quad \{\text{by (8b)}\}$$
$$h''(y) \tag{14v}$$

Here, (14r) corrects the mistake in (12l).

However, this is tantamount to requiring the user to manually write

$$\begin{array}{l}\mathbf{let}\ \hat{\mathscr{D}}_0 \overset{\triangle}{=} \mathscr{D}\,s\,0\\ \mathbf{in\ let}\ \hat{\mathscr{D}}_1 \overset{\triangle}{=} \mathscr{D}\,s\,0\\ \quad\mathbf{in}\ \hat{\mathscr{D}}_0\,(\hat{\mathscr{D}}_1\,h)\,y\end{array} \tag{15}$$

instead of:

$$\begin{array}{l}\mathbf{let}\ \hat{\mathscr{D}} \overset{\triangle}{=} \mathscr{D}\,s\,0\\ \mathbf{in}\ \hat{\mathscr{D}}\,(\hat{\mathscr{D}}\,h)\,y\end{array} \tag{16}$$

This should not be necessary since if $\mathscr{D}$ correctly implemented $\mathbb{D}$, $\hat{\mathscr{D}}_0$ and $\hat{\mathscr{D}}_1$ should be equivalent.

The essence of the bug is that the implementation of $\mathscr{D}$ in (8d) generates a distinct ϵ for each invocation $\mathscr{D}\, f\, x$, but a distinct ϵ is needed for each derivative calculation. In the first-order case, when $f : \mathbb{R} \rightarrow \mathbb{R}$, these are equivalent. Each invocation $\mathscr{D}\, f\, x$ leads to a single derivative calculation. But in the higher-order case, when f returns a function g, an invocation $\mathscr{D}\, f\, x$ yields $\bar{g}$ which performs a derivative calculation when invoked. Since $\bar{g}$ can be invoked multiple times, each such invocation will perform a distinct derivative calculation and needs a distinct ε. The implementation in the Appendix illustrates the bug when setting `*eta-expansion?*` and `*tag-substitution?*` to `#f` to use the definitions in (8d) and (8e).

7 A first solution: Eta expansion

One solution would be to eta expand the definition of $\mathscr{D}$. Such eta expansion would need to be conditional on the return type of f.

$$\mathscr{D}_1 : (\mathbb{R} \rightarrow \mathbb{R}) \rightarrow \mathbb{R} \rightarrow \mathbb{R}$$

$$\mathscr{D}_1\, f\, x_1 \stackrel{\triangle}{=} \textbf{fresh}\ \varepsilon\ \textbf{in}\ \ \textbf{tg}\ \ \varepsilon\ (f\ (x_1 + 1\varepsilon)) \tag{17a}$$

$$\mathscr{D}_2 : (\mathbb{R} \rightarrow \alpha_2 \rightarrow \mathbb{R}) \rightarrow \mathbb{R} \rightarrow \alpha_2 \rightarrow \mathbb{R}$$

$$\mathscr{D}_2\, f\, x_1\, x_2 \stackrel{\triangle}{=} \textbf{fresh}\ \varepsilon\ \textbf{in}\ \ \textbf{tg}\ \ \varepsilon\ (f\ (x_1 + 1\varepsilon)\ x_2) \tag{17b}$$

$$\mathscr{D}_3 : (\mathbb{R} \rightarrow \alpha_2 \rightarrow \alpha_3 \rightarrow \mathbb{R}) \rightarrow \mathbb{R} \rightarrow \alpha_2 \rightarrow \alpha_3 \rightarrow \mathbb{R}$$

$$\mathscr{D}_3\, f\, x_1\, x_2\, x_3 \stackrel{\triangle}{=} \textbf{fresh}\ \varepsilon\ \textbf{in}\ \ \textbf{tg}\ \ \varepsilon\ (f\ (x_1 + 1\varepsilon)\ x_2\, x_3) \tag{17c}$$

$$\vdots$$

With such eta expansion conditioned on the return type of f, (8e) is not needed, because the appropriate variant of $\mathscr{D}$ should only be invoked in a context that contains all arguments necessary to subsequently allow the call to **tg** in that invocation of $\mathscr{D}$ to yield to a non-function-containing value. This seemingly infinite set of $\mathscr{D}_i$ and associated definitions can be formulated as a single $\mathscr{D}$ with polymorphic recursion.

$$\mathscr{D}\, f\, x \stackrel{\triangle}{=} \lambda y\,.\,(\mathscr{D}\ (\lambda x\,.\,(f\, x\, y))\ x) \qquad (f\ x)\ \text{is a function} \tag{18a}$$

$$\mathscr{D}\, f\, x \stackrel{\triangle}{=} \textbf{fresh}\ \varepsilon\ \textbf{in}\ \ \textbf{tg}\ \ \varepsilon\ (f\ (x + 1\varepsilon)) \qquad (f\ x)\ \text{is not a function} \tag{18b}$$

We can see that this resolves the bug in (12a)–(12o) and accomplishes the desiderata in (14a)–(14l) without making two copies of $\hat{\mathscr{D}}$.

$$\hat{\mathscr{D}}$$

$$\Longrightarrow \qquad \{\text{by (11)}\}$$

$$\mathscr{D}\, s\, 0 \tag{19a}$$

$$\Longrightarrow \qquad \{\text{by (18a)}\}$$

$$\lambda y\,.\,(\mathscr{D}\ (\lambda x\,.\,(s\, x\, y))\ 0) \tag{19b}$$

$$\hat{\mathscr{D}}\,(\hat{\mathscr{D}}\,h)\,y$$

$\Longrightarrow$ {substitute (19b) for $\hat{\mathscr{D}}$}

$$(\lambda y\,.\,(\mathscr{D}\,(\lambda x\,.\,(s\,x\,y))\,0))\,((\lambda y\,.\,(\mathscr{D}\,(\lambda x\,.\,(s\,x\,y))\,0))\,h)\,y \tag{19c}$$

$\Longrightarrow$ {beta reduce}

$$(\lambda y\,.\,(\mathscr{D}\,(\lambda x\,.\,(s\,x\,y))\,0))\,(\mathscr{D}\,(\lambda x\,.\,(s\,x\,h))\,0)\,y \tag{19d}$$

$\Longrightarrow$ {beta reduce}

$$(\mathscr{D}\,(\lambda x\,.\,(s\,x\,(\mathscr{D}\,(\lambda x\,.\,s\,x\,h)\,0)))\,0)\,y \tag{19e}$$

$\Longrightarrow$ {by (8d)}

$$(\textbf{fresh}\,\varepsilon\,\textbf{in}\;\textbf{tg}\;\varepsilon\,((\lambda x\,.\,(s\,x\,(\mathscr{D}\,(\lambda x\,.\,(s\,x\,h))\,0)))\,(0+1\varepsilon)))\,y \tag{19f}$$

$\Longrightarrow$ {allocate a fresh tag ϵ_0}

$$(\textbf{tg}\;\epsilon_0\;((\lambda x\,.\,(s\,x\,(\mathscr{D}\,(\lambda x\,.\,(s\,x\,h))\,0)))\,(0+1\epsilon_0)))\,y \tag{19g}$$

$\Longrightarrow$ {beta reduce}

$$(\textbf{tg}\;\epsilon_0\;(s\,(0+1\epsilon_0)\,(\mathscr{D}\,(\lambda x\,.\,(s\,x\,h))\,0)))\,y \tag{19h}$$

$\Longrightarrow$ {by (8d)}

$$(\textbf{tg}\;\epsilon_0\;(s\,(0+1\epsilon_0)\,(\textbf{fresh}\,\varepsilon\,\textbf{in}\;\textbf{tg}\;\varepsilon\,((\lambda x\,.\,(s\,x\,h))\,(0+1\varepsilon)))))\,y \tag{19i}$$

$\Longrightarrow$ {allocate a fresh tag ϵ_1}

$$(\textbf{tg}\;\epsilon_0\;(s\,(0+1\epsilon_0)\,(\textbf{tg}\;\epsilon_1\;((\lambda x\,.\,(s\,x\,h))\,(0+1\epsilon_1)))))\,y \tag{19j}$$

$\Longrightarrow$ {beta reduce}

$$(\textbf{tg}\;\epsilon_0\;(s\,(0+1\epsilon_0)\,(\textbf{tg}\;\epsilon_1\;(s\,(0+1\epsilon_1)\,h))))\,y \tag{19k}$$

$\Longrightarrow$ {by (9)}

$$(\textbf{tg}\;\epsilon_0\;(s\,(0+1\epsilon_0)\,(\textbf{tg}\;\epsilon_1\;(\lambda x\,.\,(h\,(x+(0+1\epsilon_1)))))))\,y \tag{19l}$$

$\Longrightarrow$ {by (8e)}

$$(\textbf{tg}\;\epsilon_0\;(s\,(0+1\epsilon_0)\,(\textbf{tg}\;\epsilon_1)\circ(\lambda x\,.\,(h\,(x+(0+1\epsilon_1))))))\,y \tag{19m}$$

$\Longrightarrow$ {postcompose}

$$(\textbf{tg}\;\epsilon_0\;(s\,(0+1\epsilon_0)\,(\lambda x\,.\,(\textbf{tg}\;\epsilon_1\;(h\,(x+(0+1\epsilon_1)))))))\,y \tag{19n}$$

$\Longrightarrow$ {by (9)}

$$(\textbf{tg}\;\epsilon_0\;(\lambda x\,.\,((\lambda x\,.\,(\textbf{tg}\;\epsilon_1\;(h\,(x+(0+1\epsilon_1)))))\,(x+(0+1\epsilon_0)))))\,y \tag{19o}$$

$\Longrightarrow$ {beta reduce}

$$(\textbf{tg}\;\epsilon_0\;(\lambda x\,.\,(\textbf{tg}\;\epsilon_1\;(h\,((x+(0+1\epsilon_0))+(0+1\epsilon_1))))))\,y \tag{19p}$$

$\Longrightarrow$ {by (8e)}

$$(\textbf{tg}\;\epsilon_0)\circ(\lambda x\,.\,(\textbf{tg}\;\epsilon_1\;(h\,((x+(0+1\epsilon_0))+(0+1\epsilon_1)))))\,y \tag{19q}$$

$\Longrightarrow$ {postcompose}

$$(\lambda x\,.\,(\textbf{tg}\;\epsilon_0\;(\textbf{tg}\;\epsilon_1\;(h\,((x+(0+1\epsilon_0))+(0+1\epsilon_1))))))\,y \tag{19r}$$

$\Longrightarrow$ {beta reduce}

$$\textbf{tg}\;\epsilon_0\;(\textbf{tg}\;\epsilon_1\;(h\,((y+(0+1\epsilon_0))+(0+1\epsilon_1)))) \tag{19s}$$

$\Longrightarrow$ {add dual numbers}

$$\textbf{tg}\;\epsilon_0\;(\textbf{tg}\;\epsilon_1\;(h\,((y+1\epsilon_0)+(0+1\epsilon_1)))) \tag{19t}$$

$$\Longrightarrow \quad \{\text{add dual numbers}\}$$

$$\mathbf{tg}\ \epsilon_0\ (\mathbf{tg}\ \epsilon_1\ (h\,((y+1\epsilon_0)+1\epsilon_1))) \tag{19u}$$

$$\Longrightarrow \quad \{\text{same as (14r)}\}$$

$$\mathbf{tg}\ \epsilon_0\ (\mathbf{tg}\ \epsilon_1\ (h(y+1\epsilon_0)+h'(y+1\epsilon_0)\epsilon_1)) \tag{19v}$$

$$\Longrightarrow \quad \{\text{same as (14s)}\}$$

$$\mathbf{tg}\ \epsilon_0\ (\mathbf{tg}\ \epsilon_1\ ((h(y)+h'(y)\epsilon_0)+h'(y+1\epsilon_0)\epsilon_1)) \tag{19w}$$

$$\Longrightarrow \quad \{\text{same as (14t)}\}$$

$$\mathbf{tg}\ \epsilon_0\ (\mathbf{tg}\ \epsilon_1\ ((h(y)+h'(y)\epsilon_0)+(h'(y)+h''(y)\epsilon_0)\epsilon_1)) \tag{19x}$$

$$\Longrightarrow \quad \{\text{same as (14u)}\}$$

$$\mathbf{tg}\ \epsilon_0\ (h'(y)+h''(y)\epsilon_0) \tag{19y}$$

$$\Longrightarrow \quad \{\text{same as (14v)}\}$$

$$h''(y) \tag{19z}$$

Here, the allocation of a fresh tag is delayed from (19b) and is performed twice, in (19g) and (19j), allowing (19v) to correct the mistake in (12l), just like (14r). The implementation in the Appendix illustrates that this resolves the bug when setting `*eta-expansion?*` to `#t` to use the definition in (18a) and (18b) instead of that in (8d).

7.1 Issues with eta expansion

This solution presents several problems.

- First, this manuscript only considers a space of types that includes scalar reals and functions but not aggregates (exclusive of dual numbers). Complications arise when extending the space of types to include aggregates. The Appendix illustrates that the above mechanism works with functions that return Church-encoded aggregates.

$$(a,d)\ m \stackrel{\triangle}{=} m\ a\ d \tag{20a}$$

$$\mathbf{fst}\ \ c \stackrel{\triangle}{=} c\,(\lambda a\,.\,(\lambda d\,.\,a)) \tag{20b}$$

$$\mathbf{snd}\ \ c \stackrel{\triangle}{=} c\,(\lambda a\,.\,(\lambda d\,.\,d)) \tag{20c}$$

$$t\,u \stackrel{\triangle}{=} (e^{u\times u}, (\lambda f\,.\,(\lambda x\,.\,(f\,x+u)))) \tag{20d}$$

$$\mathscr{D}\,t\,1 \Longrightarrow t'(1) \tag{20e}$$

$$p \stackrel{\triangle}{=} \mathscr{D}\,t\,0 \tag{20f}$$

$$\mathbf{fst}\ p \Longrightarrow 0 \tag{20g}$$

$$\vec{\mathscr{D}} \stackrel{\triangle}{=} \mathbf{snd}\ p \tag{20h}$$

$$\vec{\mathscr{D}}\,(\vec{\mathscr{D}}\ \exp)\ 1 \Longrightarrow e \tag{20i}$$

  With a function that returned native aggregates, one would need to emulate the behavior that occurs with Church-encoded aggregates on native aggregates by delaying derivative calculation, with the associated tag allocation and **tg** applied to the native returned aggregate, until an accessor is applied to that aggregate. Consider $\mathscr{D}\,t\,0$ where $t:\mathbb{R}\rightarrow(\mathbb{R}\times((\mathbb{R}\rightarrow\mathbb{R})\rightarrow\mathbb{R}))$ as above. One could not perform the derivative calculation when computing the value p returned by $\mathscr{D}\,t\,0$. One would have to delay until applying an accessor to p. If one accessed the first element of p, one would perform the derivative calculation, with the associated tag allocation, at the time of access. But if one accessed the second element of p, one would have to further delay the derivative calculation, with the associated tag allocation, until that second element was invoked. This could require different amounts of delay that might be incompatible with some static type systems.

- Second, with a type system or other static analysis mechanism that is unable to handle the unbounded polymorphism of (17a), (17b), (17c), ... or infer the "is [not] a function" side conditions of (18a) and (18b), achieving completeness might require run-time evaluation of the side conditions. This could involve calling f twice, once to determine its return type and once to do the eta-expanded derivative calculation, and lead to exponential increase in asymptotic time complexity.
- Third, the solution can break sharing in curried functions, even with a type system or other static analysis mechanism that is able to eliminate the run-time evaluation of "is [not] a function" side conditions. Consider

$$g\,x \triangleq \mathbf{let}\, t \triangleq f\,x\, \mathbf{in}\, \lambda p\,.\,p\,t \tag{21}$$

invoked in:

$$h\,x \triangleq \mathbf{let}\, c \triangleq g\,x\, \mathbf{in}\, (c\,(\lambda t\,.\,t)) + (c\,(\lambda t\,.\,(\lambda u\,.\,t \times u))\,\pi) \tag{22}$$

The programmer would expect $h\,8$ to call f once in the calculation of the temporary $t = f\,8$. And indeed this is what would occur in practice. Now consider $\mathscr{D}\,h\,8$. The strategy discussed above would (in the absence of memoization or similar heroic measures) end up calculating $f\,8$ twice, as the delayed tag allocation would end up splitting into two independent tag allocations with each independently redoing the calculation. This violates the constant-factor-overhead complexity guarantee of forward AD, imposing, in the worst case, exponential overhead.

8 A second solution: Tag substitution

Another solution would be to wrap $\bar{g}$ with tag substitution to guard against tag collision, replacing (8e) with:

$$\mathbf{tg}\;\varepsilon_1\,\bar{g}\,y \triangleq \mathbf{fresh}\,\varepsilon\,\mathbf{in}\,([\varepsilon_1/\varepsilon] \circ (\mathbf{tg}\;\varepsilon_1) \circ \bar{g} \circ [\varepsilon/\varepsilon_1])\,y \qquad \bar{g} \text{ is a function} \tag{23}$$

Here $[\varepsilon_1/\varepsilon_2]\,x$ substitutes ε_1 for ε_2 in x. In a language with opaque closures, tag substitution must operate on functions by appropriate pre- and post-composition.

$$[\varepsilon_1/\varepsilon_2]\,a \triangleq a \qquad a : \mathbb{R} \tag{24a}$$
$$[\varepsilon_1/\varepsilon_2]\,(a + b\varepsilon_2) \triangleq a + b\varepsilon_1 \tag{24b}$$
$$[\varepsilon_1/\varepsilon_2]\,(a + b\varepsilon) \triangleq ([\varepsilon_1/\varepsilon_2]\,a) + ([\varepsilon_1/\varepsilon_2]\,b)\varepsilon \qquad \varepsilon \neq \varepsilon_2 \tag{24c}$$
$$[\varepsilon_1/\varepsilon_2]\,\bar{g}\,y \triangleq \mathbf{fresh}\,\varepsilon\,\mathbf{in}\;\;([\varepsilon_2/\varepsilon] \circ [\varepsilon_1/\varepsilon_2] \circ \bar{g} \circ [\varepsilon/\varepsilon_2])\,y \qquad \bar{g} \text{ is a function} \tag{24d}$$

The intent of (24d) is to substitute ε_1 for ε_2 in values closed-over in $\bar{g}$. An ε_2 in the output of $\bar{g}$ can result either from closed-over values and/or input values. We want to substitute for instances of ε_2 in the output that result from the former but not the latter. This is accomplished by substituting a fresh tag for instances of ε_2 in the input and substituting them back at the output to preserve the extensional behavior of $\bar{g}$. Equation (23) operates in a similar fashion. The intent of (23) is to extract the coefficient of instances of ε_1 in the output of $\bar{g}$ that result from closed-over values, not input values. This is accomplished by substituting a fresh tag for instances of ε_1 in the input and substituting them back at the output to preserve the extensional behavior of $\bar{g}$.

We can see that this also resolves the bug in (12a)–(12o) and accomplishes the desiderata in (14a)–(14l) without making two copies of $\hat{\mathscr{D}}$.

$$\hat{\mathscr{D}}$$

$$\Longrightarrow \quad \{\text{by (11)}\}$$

$$\mathscr{D}\, s\, 0 \tag{25a}$$

$$\Longrightarrow \quad \{\text{by (8d)}\}$$

$$\textbf{fresh}\ \varepsilon\ \textbf{in}\ \ \textbf{tg}\ \ \varepsilon\ (s\ (0+1\varepsilon)) \tag{25b}$$

$$\Longrightarrow \quad \{\text{allocate a fresh tag } \epsilon_0\}$$

$$\textbf{tg}\ \ \epsilon_0\ \ (s\ (0+1\epsilon_0)) \tag{25c}$$

$$\Longrightarrow \quad \{\text{by (9)}\}$$

$$\textbf{tg}\ \ \epsilon_0\ \ (\lambda f\,.\,\lambda x\,.\,(f\ (x+1\epsilon_0))) \tag{25d}$$

$$\Longrightarrow \quad \{\text{by (23)}\}$$

$$\lambda y\,.\,(\textbf{fresh}\ \varepsilon\ \textbf{in}\ ([\epsilon_0/\varepsilon]\circ(\textbf{tg}\ \ \epsilon_0)\circ(\lambda f\,.\,\lambda x\,.\,(f\ (x+1\epsilon_0)))\circ[\varepsilon/\epsilon_0])\ y) \tag{25e}$$

$$\hat{\mathscr{D}}\ (\hat{\mathscr{D}}\ h)\ y$$

$$\Longrightarrow \quad \{\text{substitute (25e) for } \hat{\mathscr{D}}\}$$

$$\begin{array}{l}\lambda y\,.\,(\textbf{fresh}\ \varepsilon\ \textbf{in}\ ([\epsilon_0/\varepsilon]\circ(\textbf{tg}\ \ \epsilon_0)\circ(\lambda f\,.\,\lambda x\,.\,(f\ (x+1\epsilon_0)))\circ[\varepsilon/\epsilon_0])\ y)\\ (\lambda y\,.\,(\textbf{fresh}\ \varepsilon\ \textbf{in}\ ([\epsilon_0/\varepsilon]\circ(\textbf{tg}\ \ \epsilon_0)\circ(\lambda f\,.\,\lambda x\,.\,(f\ (x+1\epsilon_0)))\circ[\varepsilon/\epsilon_0])\ y)\ h)\\ y\end{array} \tag{25f}$$

$$\Longrightarrow \quad \{\text{beta reduce}\}$$

$$\begin{array}{l}\lambda y\,.\,(\textbf{fresh}\ \varepsilon\ \textbf{in}\ ([\epsilon_0/\varepsilon]\circ(\textbf{tg}\ \ \epsilon_0)\circ(\lambda f\,.\,\lambda x\,.\,(f\ (x+1\epsilon_0)))\circ[\varepsilon/\epsilon_0])\ y)\\ (\textbf{fresh}\ \varepsilon\ \textbf{in}\ ([\epsilon_0/\varepsilon]\circ(\textbf{tg}\ \ \epsilon_0)\circ(\lambda f\,.\,\lambda x\,.\,(f\ (x+1\epsilon_0)))\circ[\varepsilon/\epsilon_0])\ h)\\ y\end{array} \tag{25g}$$

$$\Longrightarrow \quad \{\text{beta reduce}\}$$

$$\begin{array}{l}(\textbf{fresh}\ \varepsilon\ \textbf{in}\ ([\epsilon_0/\varepsilon]\circ(\textbf{tg}\ \ \epsilon_0)\circ(\lambda f\,.\,\lambda x\,.\,(f\ (x+1\epsilon_0)))\circ[\varepsilon/\epsilon_0])\\ (\textbf{fresh}\ \varepsilon\ \textbf{in}\ ([\epsilon_0/\varepsilon]\circ(\textbf{tg}\ \ \epsilon_0)\circ(\lambda f\,.\,\lambda x\,.\,(f\ (x+1\epsilon_0)))\circ[\varepsilon/\epsilon_0])\ h))\\ y\end{array} \tag{25h}$$

$$\Longrightarrow \quad \{\text{allocate a fresh tag } \epsilon_1\}$$

$$\begin{array}{l}(([\epsilon_0/\epsilon_1]\circ(\textbf{tg}\ \ \epsilon_0)\circ(\lambda f\,.\,\lambda x\,.\,(f\ (x+1\epsilon_0)))\circ[\epsilon_1/\epsilon_0])\\ \ (\textbf{fresh}\ \varepsilon\ \textbf{in}\ ([\epsilon_0/\varepsilon]\circ(\textbf{tg}\ \ \epsilon_0)\circ(\lambda f\,.\,\lambda x\,.\,(f\ (x+1\epsilon_0)))\circ[\varepsilon/\epsilon_0])\ h))\\ y\end{array} \tag{25i}$$

$$\Longrightarrow \quad \{\text{allocate a fresh tag } \epsilon_2\}$$

$$\begin{array}{l}(([\epsilon_0/\epsilon_1]\circ(\textbf{tg}\ \ \epsilon_0)\circ(\lambda f\,.\,\lambda x\,.\,(f\ (x+1\epsilon_0)))\circ[\epsilon_1/\epsilon_0])\\ \ (([\epsilon_0/\epsilon_2]\circ(\textbf{tg}\ \ \epsilon_0)\circ(\lambda f\,.\,\lambda x\,.\,(f\ (x+1\epsilon_0)))\circ[\epsilon_2/\epsilon_0])\ h))\\ y\end{array} \tag{25j}$$

$\Longrightarrow$ {substitute ϵ_2 for ϵ_0, which leaves h unchanged since it can't close over the freshly allocated tags}

$$((([\epsilon_0/\epsilon_1] \circ (\textbf{tg}\ \epsilon_0) \circ (\lambda f\,.\,\lambda x\,.\,(f\ (x+1\epsilon_0))) \circ [\epsilon_1/\epsilon_0]) \\ (([\epsilon_0/\epsilon_2] \circ (\textbf{tg}\ \epsilon_0) \circ (\lambda f\,.\,\lambda x\,.\,(f\ (x+1\epsilon_0)))) \, h)) \\ y \tag{25k}$$

$\Longrightarrow$ {beta reduce and postcompose}

$$((([\epsilon_0/\epsilon_1] \circ (\textbf{tg}\ \epsilon_0) \circ (\lambda f\,.\,\lambda x\,.\,(f\ (x+1\epsilon_0))) \circ [\epsilon_1/\epsilon_0]) \\ (\lambda x\,.\,([\epsilon_0/\epsilon_2]\ (\textbf{tg}\ \epsilon_0\ \ (h\ (x+1\epsilon_0)))))) \\ y \tag{25l}$$

$\Longrightarrow$ {substitute ϵ_1 for ϵ_0}

$$((([\epsilon_0/\epsilon_1] \circ (\textbf{tg}\ \epsilon_0) \circ (\lambda f\,.\,\lambda x\,.\,(f\ (x+1\epsilon_0)))) \\ (\lambda x\,.\,([\epsilon_1/\epsilon_2]\ (\textbf{tg}\ \epsilon_1\ \ (h\ (x+1\epsilon_1)))))) \\ y \tag{25m}$$

$\Longrightarrow$ {beta reduce and postcompose}

$$(\lambda x\,.\,([\epsilon_0/\epsilon_1]\ (\textbf{tg}\ \epsilon_0\ \ ((\lambda x\,.\,([\epsilon_1/\epsilon_2]\ (\textbf{tg}\ \epsilon_1\ \ (h\ (x+1\epsilon_1)))))\ (x+1\epsilon_0)))))\ y \tag{25n}$$

$\Longrightarrow$ {beta reduce}

$$[\epsilon_0/\epsilon_1]\ (\textbf{tg}\ \epsilon_0\ \ ((\lambda x\,.\,([\epsilon_1/\epsilon_2]\ (\textbf{tg}\ \epsilon_1\ \ (h\ (x+1\epsilon_1)))))\ (y+1\epsilon_0))) \tag{25o}$$

$\Longrightarrow$ {beta reduce}

$$[\epsilon_0/\epsilon_1]\ (\textbf{tg}\ \epsilon_0\ \ ([\epsilon_1/\epsilon_2]\ (\textbf{tg}\ \epsilon_1\ \ (h\ ((y+1\epsilon_0)+1\epsilon_1))))) \tag{25p}$$

$\Longrightarrow$ {apply h to a dual number}

$$[\epsilon_0/\epsilon_1]\ (\textbf{tg}\ \epsilon_0\ \ ([\epsilon_1/\epsilon_2]\ (\textbf{tg}\ \epsilon_1\ \ (h(y+1\epsilon_0)+h'(y+1\epsilon_0)\epsilon_1)))) \tag{25q}$$

$\Longrightarrow$ {apply h to a dual number}

$$[\epsilon_0/\epsilon_1]\ (\textbf{tg}\ \epsilon_0\ \ ([\epsilon_1/\epsilon_2]\ (\textbf{tg}\ \epsilon_1\ \ ((h(y)+h'(y)\epsilon_0)+h'(y+1\epsilon_0)\epsilon_1)))) \tag{25r}$$

$\Longrightarrow$ {apply h to a dual number}

$$[\epsilon_0/\epsilon_1]\ (\textbf{tg}\ \epsilon_0\ \ ([\epsilon_1/\epsilon_2]\ (\textbf{tg}\ \epsilon_1\ \ ((h(y)+h'(y)\epsilon_0)+(h'(y)+h''(y)\epsilon_0)\epsilon_1)))) \tag{25s}$$

$\Longrightarrow$ {by (8b)}

$$[\epsilon_0/\epsilon_1]\ (\textbf{tg}\ \epsilon_0\ \ ([\epsilon_1/\epsilon_2]\ (h'(y)+h''(y)\epsilon_0))) \tag{25t}$$

$\Longrightarrow$ {substitute ϵ_1 for ϵ_2}

$$[\epsilon_0/\epsilon_1]\ (\textbf{tg}\ \epsilon_0\ \ (h'(y)+h''(y)\epsilon_0)) \tag{25u}$$

$\Longrightarrow$ {by (8b)}

$$[\epsilon_0/\epsilon_1]\ h''(y) \tag{25v}$$

$\Longrightarrow$ {substitute ϵ_0 for ϵ_1}

$$h''(y) \tag{25w}$$

Steps (25k) and (25m) are abbreviated as they really use (24d). Here, the tag substitution in (25m) allows (25q) to correct the mistake in (12l), just like (14r). The implementation in the Appendix illustrates that this resolves the bug when setting `*tag-substitution?*` to `#t` to use the definition in (23) instead of that in (8e).

8.1 Issues with tag substitution

This solution presents several problems, when implemented as user code in a pure language. In the presence of aggregates, unless care is taken, the computational burden of tag substitution can violate the complexity guarantees of forward AD. The call to **tg** in step 3 might take longer than unit time as tag substitution must potentially traverse an aggregate of arbitrary size. When that aggregate shares substructure, a careless implementation might traverse such shared substructure multiple times, leading to potential exponential growth in time complexity. Moreover, a careless implementation might copy shared substructure multiple times, leading to potential exponential growth in space complexity. Laziness, memoization, and hash-consing might solve this, but it can be tricky to employ such in a fashion that preserves the requisite time and space complexity guarantees of forward AD, particularly in a pure or multithreaded context.

We are unsure, however, that laziness, memoization, and hash-consing completely eliminate the problem. First, some languages like PYTHON and SCHEME lack the requisite pervasive default laziness. Failure to explicitly code the correct portions of user code as lazy in an eager language can break the complexity guarantees in subtle ways. But there are subtle issues even in languages like HASKELL with the requisite pervasive default laziness, and even when laziness is correctly introduced manually in eager languages. One is that memoization and hash-consing implicitly involve a notion of equality. But it is not clear what notion of equality to use, especially with "gensym" and potential alpha equivalence. One might need `eq?`, pointer or intensional equivalence, rather than `equal?`, structural or extensional equivalence, and all of the impurity that this introduces. Further, memoization and hash-consing might themselves be a source of a new kind of perturbation confusion if tags can persist. One would then need to substitute the memoized tags or the hash-cons cache. Beyond this, memoization and hash-consing could break space complexity guarantees unless the cache were flushed. It is not clear when/where to flush the cache, and even whether there is a consistent place to do so. There might be inconsistent competing concerns. Finally, many systems don't provide the requisite hooks to do all of this. One would need weak pointers and finalization. All of this deserves further investigation.

The above difficulties only arise when implementing tag substitution as user code in a pure language. The opacity of closures necessitates implementing tag substitution on functions via pre- and post-composition (24d). The complexity guarantees of forward AD could be maintained if the substitution mechanism $[\varepsilon_1/\varepsilon_2]\,x$ were implemented so that it

(a) did not traverse shared substructure multiple times,
(b) copied shared substructure during renaming in a fashion that preserved structure sharing, and
(c) could apply to closures, by accessing, copying, renaming, and reclosing around the environments inside closures, without resorting to pre- and post-composition.

This could be accomplished either by including the $[\varepsilon_1/\varepsilon_2]\,x$ mechanism as a primitive in the implementation, or by providing other lower-level primitives out of which it could be fashioned. One such mechanism is `map-closure`, the ability to reflectively access and modify closure environments (Siskind & Pearlmutter, 2007).

9 Differential geometry and the push forward operator

The definition (3) only extends $\mathbb{D}$, and the mechanisms of Sections 7 and 8 only extend $\mathscr{D}$, to higher-order functions $\mathbb{R} \to \alpha$ whose ranges are functions. Differential geometry provides the framework for extending $\mathbb{D}$ to functions $\alpha_1 \to \alpha_2$ whose domains too are functions.

Differential geometry concerns itself with differentiable mappings between manifolds, where intuitively a manifold is a surface along which points can move smoothly, like the surface of a sphere or the space of $n \times n$ rotation matrices. Given a point x, called a *primal* (value), on a manifold α, we can consider infinitesimal perturbations of x. The space of such perturbations is a vector space called a *tangent space*, denoted by $T_x\alpha$. This is a dependent type, dependent on the primal x. A particular perturbation, an element x' of the tangent space, is called a *tangent* (value). A pair (x, x') of a primal and tangent value is called a *bundle* (value), which are members of a bundle space $T\alpha = \sum_{x:\alpha} \{x\} \times T_x\alpha$. Bundles generalize the notion of dual numbers. So if x has type α, for some α, the tangent x' has type $T_x\alpha$, and they can be bundled together as $(x + x'\epsilon)$ which has type $T\alpha$.

The machinery of differential geometry defines $T_x\alpha$ for various manifolds and spaces α. For function spaces $\alpha \to \beta$, where f is of type $\alpha \to \beta$, $T_f(\alpha \to \beta) = (a : \alpha) \to T_{f(a)}\beta$ and $T(\alpha \to \beta) = \alpha \to T\beta$. The function **bundle** $(x : \alpha)\ (x' : T_x\alpha) \mapsto (x, x') : T\alpha$ constructs a bundle from a primal and a tangent, and the function **tangent** $(x, x') : T\alpha \mapsto x' : T_x\alpha$ extracts a tangent from a bundle. Differential geometry provides a *push forward* operator that generalizes the notion of a univariate derivative from functions f of type $\mathbb{R} \to \mathbb{R}$ to functions f of type $\alpha \to \beta$.

$$\textbf{pf} : (\alpha \to \beta) \to (T\alpha \to T\beta) \tag{26}$$

This augments the original mapping $(a : \alpha) \to \beta$ to also *linearly* map a tangent $T_a\alpha$ of the input a to a tangent $T_{f(a)}\beta$ of the output $f(a)$.

Here we sketch how to materialize differential geometry as program constructs to generalize $\mathbb{D}$ to functions $\alpha_1 \to \alpha_2$ whose domains (and ranges) are functions. A full treatment is left for future work. We first note that:

$$\mathbb{D}\, f\, x = \textbf{tangent}\,(\textbf{pf}\, f\, (\textbf{bundle}\ x\, 1)) \tag{27}$$

This only applies when $x : \mathbb{R}$ because of the constant 1. We can generalize this to a directional derivative:

$$\overrightarrow{\mathbb{J}}\, f\, x\, x' = \textbf{tangent}\,(\textbf{pf}\, f\, (\textbf{bundle}\ x\, x')) \tag{28}$$

This further generalizes to x of any type. With this, $\mathbb{D}$ becomes a special case of $\overrightarrow{\mathbb{J}}$:

$$\mathbb{D}\, f\, x = \overrightarrow{\mathbb{J}}\, f\, x\, 1 \tag{29}$$

To materialize $\overrightarrow{\mathbb{J}}$ in (28), we need to materialize **tangent**, **pf**, and **bundle**. The definition of **tg** in (8a)–(8c) and (8e) materializes **tangent** with the first solution, eta expansion (Section 7), while that in (8a)–(8c) and (23) does so with the second solution, tag substitution (Section 8). The nonstandard interpretation of the arithmetic basis functions sketched in (5a) and (5b) materializes **pf** by lifting a computation on real numbers to a computation on dual numbers. All that remains is to materialize **bundle**. So far, we have been simply writing this as step 2, a map from a to $a + 1\epsilon$ or a map from x to $x + 1\varepsilon$ in (8d). This only works for numbers, not functions. With the framework of the first solution, eta expansion (Section 7), we can extend this to functions:

$$\mathbf{bun}\ \varepsilon\, x\, x' \triangleq x + x'\varepsilon \qquad x \text{ and } x' \text{ are not functions} \tag{30a}$$

$$\mathbf{bun}\ \varepsilon\, f\, f'\, y \triangleq \mathbf{bun}\ \varepsilon\ (f\ y)\ (f'\ y) \qquad f \text{ and } f' \text{ are functions} \tag{30b}$$

Recalling footnote 2, the post-composition in (30b) is analogous to that in (8e). With the framework of the second solution, tag substitution (Section 8), we would need the alternative:

$$\begin{aligned}\mathbf{bun}\ \varepsilon_1\, f\, f'\, y \triangleq{} & \mathbf{fresh}\ \varepsilon \\ & \mathbf{in}\ [\varepsilon_1/\varepsilon]\ (\mathbf{bun}\ \varepsilon_1\ (f\ ([\varepsilon/\varepsilon_1]\, y))\ (f'\ ([\varepsilon/\varepsilon_1]\, y)))\end{aligned} \qquad f \text{ and } f' \text{ are functions} \tag{31}$$

to (30b). The additional tag substitution in (31) is analogous to that in (23). With this, we can now materialize $\overrightarrow{\mathbb{J}}$ in the framework of the first solution, eta expansion (Section 7):

$$\overrightarrow{\mathcal{J}}\, f\, x\, x' \triangleq \lambda y\,.\,(\overrightarrow{\mathcal{J}}\ (\lambda x\,.\,(f\, x\, y))\, x\, x') \qquad (f\ x) \text{ is a function} \tag{32a}$$

$$\overrightarrow{\mathcal{J}}\, f\, x\, x' \triangleq \mathbf{fresh}\ \varepsilon\ \mathbf{in}\ \mathbf{tg}\ \varepsilon\ (f\ (\mathbf{bun}\ \varepsilon\, x\, x')) \qquad (f\ x) \text{ is not a function} \tag{32b}$$

which is analogous to (18a) and (18b), and in the framework of the second solution, tag substitution (Section 8):

$$\overrightarrow{\mathcal{J}}\, f\, x\, x' \triangleq \mathbf{fresh}\ \varepsilon\ \mathbf{in}\ \mathbf{tg}\ \varepsilon\ (f\ (\mathbf{bun}\ \varepsilon\, x\, x')) \tag{33}$$

which is analogous to (8d). With this, $\mathscr{D}$ becomes a special case of $\overrightarrow{\mathcal{J}}$:

$$\mathscr{D}\, f\, x \triangleq \overrightarrow{\mathcal{J}}\, f\, x\, 1 \tag{34}$$

The implementation in the Appendix illustrates this when setting `*section9?*` to `#t` to use (34) instead of either (18a) and (18b) or (8d). Moreover, the implementation in the Appendix illustrates that:

$$\mathbf{mapPair}\ f\, l \triangleq (f\ (\mathbf{fst}\ l)), (f\ (\mathbf{snd}\ l)) \tag{35a}$$

$$\mathbf{sqr}\ x \triangleq x \times x \tag{35b}$$

$$\overrightarrow{\mathcal{J}}\ \mathbf{mapPair}\ \mathbf{sqr}\ (\mathscr{D}\ \mathbf{sqr})\ (5, 10) \Longrightarrow (10, 20) \tag{35c}$$

There is a crucial difference, however, between **bundle** and **tangent** and the corresponding materializations **bun** and **tg**. The former do not take ε as an argument. This allows them to be used as distinct notational entities. In contrast, **bun** and **tg** must take the *same* ε as an argument, this tag *must* be fresh, and it should not be used anywhere else. Thus it should not escape, except in ways that are protected by tag substitution. This motivates creation of the $\overrightarrow{\mathcal{J}}$ construct. There is no corresponding standard $\overrightarrow{\mathbb{J}}$ construct in differential geometry; we created it just to describe the intended meaning of $\overrightarrow{\mathcal{J}}$.

This generalization still suffers from the poor complexity properties in Sections 7.1 and 8.1. We do not know how to provide a materialization of differential geometry or a program construct that can take derivatives of higher-order functions whose domains and/or ranges include (higher order) functions in a fashion that exhibits the complexity guarantees of forward AD. Moreover, we don't even know whether it is possible.

10 Conclusion

Classical AD systems, such as ADIFOR (Bischof *et al.*, 1992), TAPENADE (Hascoët & Pascual, 2004, and FADBAD++ (Bendtsen & Stauning, 1996), were implemented for first-order languages like FORTRAN, C, and C++. This made it difficult to formulate situations like (7) where the kind of perturbation confusion reported by Siskind & Pearlmutter (2005) can arise. Thus classical AD systems did not implement the tagging mechanisms reported by Pearlmutter & Siskind (2007) and Siskind & Pearlmutter (2008). Moreover, such classical AD systems do not expose a derivative-taking operator as a higher-order function, let alone one that can take derivatives of higher-order functions. In these systems, it is difficult to formulate the bug in Section 5.

Note that the difficulty arises from the nature of the language whose code is differentiated and not the fact that many classical systems like ADIFOR and TAPENADE expose AD to the user via a source-code transformation implemented via a preprocessor rather than a higher-order function. Conceptually, both a higher-order function and a preprocessor applying a transformation to source code map functions to functions. Thus while one might write:

$$\begin{aligned}&\textbf{let } f' \triangleq \mathscr{D}\, f\\ &\textbf{in } \ldots f'(x)\ldots\end{aligned} \tag{36}$$

in a system that exposes AD to the user with an interface as a higher-order function $\mathscr{D}$, one would accomplish essentially the same thing in a system that exposes AD to the user with a preprocessor that implements a source-code transformation by having the preprocessor compute the let binding $f' \triangleq \mathscr{D}\, f$. The issue presented in this manuscript would arise even in a framework that exposes AD to the user with a preprocessor that implements a source-code transformation if one would write

$$\begin{aligned}&\textbf{let } s' \triangleq \mathscr{D}\, s\\ &\textbf{in let } \hat{\mathscr{D}} \triangleq s'\ 0\\ &\quad\textbf{in } \hat{\mathscr{D}}\ (\hat{\mathscr{D}}\ h)\ y\end{aligned} \tag{37}$$

and have the preprocessor compute the let binding $s' \triangleq \mathscr{D}\, s$. The difficulty in formulating the issue presented in this manuscript follows from the fact that classical languages like FORTRAN, C, and C++ lack the capacity for higher-order functions (closures) needed to perform the let binding $\hat{\mathscr{D}} \triangleq s'\ 0$, not from any aspect of the difference between exposing AD via an interface via a higher-order function versus a preprocessor that implements a source-code transformation. Indeed, the issue described here would manifest in a system that exposed AD via a preprocessor that implements a source-code transformation in a language such as PYTHON that supports the requisite closures and higher-order functions (e.g., MYIA, Breuleux & van Merriënboer, 2017 and TANGENT, van Merriënboer *et al.*, 2018).

Recent AD systems, such as MYIA, TANGENT, and those in footnote 1, as well as the HASKELL AD package available on Cabal (Kmett, 2010), the "Beautiful Differentiation" system (Elliott, 2009), and the "Compiling to Categories" system (Elliott, 2017), have been implemented for higher-order languages like SCHEME, ML, HASKELL, F♯, PYTHON, LUA, and JULIA. One by one, many of these systems have come to discover the kind of perturbation confusion reported by Siskind & Pearlmutter (2005) and have come to implement the tagging mechanisms reported by Pearlmutter & Siskind (2007) and Siskind & Pearlmutter (2008). Moreover, all these recent systems expose a derivative-taking operator as a higher-order function. However, except for SCMUTILS, none supported taking derivatives of higher-order functions.

Prior to its 30 August 2011 release, SCMUTILS, the only forward AD system that supported taking derivatives of higher-order functions, employed the mechanism of (8a)–(8e) and exhibited the bug in Section 5. An attempt was made to fix this bug in the 30 August 2011 release of SCMUTILS, using the second solution, tag substitution, discussed in Section 8, in response to an early version of this manuscript. SCMUTILS was patched to include code that is similar to, but not identical to, (23) and (24a)–(24d). Crucially, it allocates a fresh tag in its implementation of (23) but not in its implementation of (24d); its implementation of (24d) being

$$[\varepsilon_1/\varepsilon_2]\,\bar{g} \triangleq [\varepsilon_2/\varepsilon_1]\circ\bar{g}\circ[\varepsilon_1/\varepsilon_2]. \qquad \bar{g}\text{ is a function} \tag{38}$$

This, however, is incorrect, as illustrated by the following variant of the bug in Section 5:

$$v\,u\,f_1\,f_2\,x \triangleq f_1\,f_2\,(x+u) \tag{39}$$

$$i\,x \triangleq x \tag{40}$$

Variants of (10a)–(10c) show that $\mathbb{D}\,v\,0\,(\mathbb{D}\,v\,0\,i)\,h\,y = h''(y)$. The 27 August 2016 release, the current release at the time of writing, however, yields $\mathscr{D}\,v\,0\,(\mathscr{D}\,v\,0\,i)\,h\,y \Longrightarrow 0$. Both solutions presented here yield the correct result.

In 2019, the authors reached out to Gerald Jay Sussman, one of the authors of SCMUTILS, to help fix SCMUTILS. He asked whether we could produce an example that illustrated the necessity of performing substitution on functions (24d) and why an alternate

$$[\varepsilon_1/\varepsilon_2]\,\bar{g} \triangleq \bar{g} \qquad \bar{g}\text{ is a function} \tag{41}$$

that did not perform substitution on functions wouldn't suffice. A variant of (9) and (11) that wraps and unwraps arguments and results in Church-encoded boxes illustrates the necessity of (24d).

$$\begin{aligned}&\textsc{box} : \mathbb{R}\to\square\mathbb{R}\\ &\textsc{box}\,x\,m \triangleq m\,x\end{aligned} \tag{42a}$$

$$\begin{aligned}&\textsc{unbox} : \square\mathbb{R}\to\mathbb{R}\\ &\textsc{unbox}\,x \triangleq x\,(\lambda x\,.\,x)\end{aligned} \tag{42b}$$

$$\begin{aligned}&\textsc{wrap} : (\mathbb{R}\to\mathbb{R})\to(\square\mathbb{R}\to\square\mathbb{R})\\ &\textsc{wrap}\,f\,x \triangleq \textsc{box}\,(f\,(\textsc{unbox}\,x))\end{aligned} \tag{42c}$$

$$\begin{aligned}&\textsc{unwrap} : (\square\mathbb{R}\to\square\mathbb{R})\to(\mathbb{R}\to\mathbb{R})\\ &\textsc{unwrap}\,f\,x \triangleq \textsc{unbox}\,(f\,(\textsc{box}\,x))\end{aligned} \tag{42d}$$

$$\begin{aligned}&\textsc{wrapTwo} : ((\mathbb{R}\to\mathbb{R})\to(\mathbb{R}\to\mathbb{R}))\to((\square\mathbb{R}\to\square\mathbb{R})\to(\square\mathbb{R}\to\square\mathbb{R}))\\ &\textsc{wrapTwo}\,f\,g\,x \triangleq \textsc{box}\,((f\,(\textsc{unwrap}\,g))\,(\textsc{unbox}\,x))\end{aligned} \tag{42e}$$

$$\textsc{wrapTwoResult} : (\mathbb{R} \to ((\mathbb{R} \to \mathbb{R}) \to (\mathbb{R} \to \mathbb{R}))) \to (\mathbb{R} \to ((\Box\,\mathbb{R} \to \Box\,\mathbb{R}) \to (\Box\,\mathbb{R} \to \Box\,\mathbb{R})))$$
$$\textsc{wrapTwoResult}\, f\, x \triangleq \textsc{wrapTwo}\ \ (f\, x) \tag{42f}$$

$$\textsc{wrapped}\hat{\mathscr{D}} \triangleq \mathscr{D}\, (\textsc{wrapTwoResult}\, s)\; 0 \tag{42g}$$

The same analysis as (10a)–(10c) shows that:

$$\textsc{unwrap}\, (\mathbb{D}\, (\textsc{wrapTwoResult}\, s)\; 0\, (\mathbb{D}\, (\textsc{wrapTwoResult}\, s)\; 0\, (\textsc{wrap}\, h))) = h'' \tag{42h}$$

While

$$\textsc{unwrap}\, (\mathscr{D}\, (\textsc{wrapTwoResult}\, s)\; 0\, (\mathscr{D}\, (\textsc{wrapTwoResult}\, s)\; 0\, (\textsc{wrap}\, h))) = h'' \tag{42i}$$

with both (24d) and (41), with (24d),

$$\textsc{unwrap}\, (\textsc{wrapped}\hat{\mathscr{D}}\, (\textsc{wrapped}\hat{\mathscr{D}}\, (\textsc{wrap}\, h))) = h'' \tag{42j}$$

but with (41),

$$\textsc{unwrap}\, (\textsc{wrapped}\hat{\mathscr{D}}\, (\textsc{wrapped}\hat{\mathscr{D}}\, (\textsc{wrap}\, h))) \neq h'' \tag{42k}$$

The authors of SCMUTILS are in the process of fixing it again in response to this updated manuscript. The tenacity of this bug illustrates its subtlety and cries out for a proof of correctness.

Practically all systems that expose a derivative-taking operator as a higher-order function generalize that operator to take gradients and Jacobians of functions whose domains and/or ranges are aggregates, and most have come to implement tagging. The current forefront of deep learning research often involves nested application of AD and application of AD to higher-order functions (Maclaurin *et al.*, 2015b; Andrychowicz *et al.*, 2016; Raissi, 2018; Chen *et al.*, 2018; Salman *et al.*, 2018). This work often combines building custom frameworks to support the particular derivatives of interest, and performing transformations (closure conversion or even full AD transforms) manually. Under the pressure of machine learning programmers' desire for nesting and for derivatives of higher-order functions, it is reasonable to speculate that many, if not most, of the above systems will attempt to support these usage patterns. We hope that the awareness provided by this manuscript will help such efforts avoid this particular subtle bug.

Without formal proofs, we cannot really be sure whether the first solution, eta expansion ((8a)–(8c), (18a), (18b)), or the second solution, tag substitution ((8a)–(8d), (23)), correctly implements the specification in (3). We cannot even be sure that (8a)–(8d) correctly implement the specification in (1). These are tricky due to subtleties like nondifferentiability, nontermination, and the difference between function intensions and extensions pointed out by Siskind & Pearlmutter (2008, footnote 1). Ehrhard & Regnier (2003), Manzyuk (2012a, 2012b), Kelly *et al.* (2016), and Plotkin (2018) present promising work in this direction. Given these sorts of subtle bugs, and the growing interest in—and economic and societal importance of—complicated software systems driven by nested automatically calculated derivatives, it is our hope that formal methods can bridge the gap between the Calculus and the Lambda Calculus, allowing derivatives of interest of arbitrary programs to be not just automatically and efficiently calculated, but also for their correctness to be formally verified.

Acknowledgments

We would like to thank Gerald Jay Sussman for wrestling at our side in the search for correctness. We also appreciate the efforts of Olivier Danvy in helping improve an earlier version of this manuscript, and Jeremy Gibbons for helping to improve this version. This work was supported, in part, by Science Foundation Ireland (SFI) Principal Investigator grant 09/IN.1/I2637, by the Army Research Laboratory (ARL) accomplished under Cooperative Agreement Number W911NF-10-2-0060, by the National Science Foundation (NSF) under Grants 1522954-IIS and 1734938-IIS, and by the Intelligence Advanced Research Projects Activity (IARPA) via Department of Interior/Interior Business Center (DOI/IBC) contract number D17PC00341. Any opinions, findings, views, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views, official policies, or endorsements, either expressed or implied, of SFI, ARL, NSF, IARPA, DOI/IBC, or the Irish or the U.S. Governments. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes, notwithstanding any copyright notation herein.

Supplementary material

To view supplementary material for this article, please visit `http://dx.doi.org/10.1017/S095679681900008X`.

References

Andrychowicz, M., Denil, M., Colmenarejo, S. G., Hoffman, M. W., Pfau, D., Schaul, T. & de Freitas, N. (2016) Learning to learn by gradient descent by gradient descent. In *Advances in Neural Information Processing Systems*. Red Hook, NY, USA: Curran Associates.

Baydin, A. G., Pearlmutter, B. A. & Siskind, J. M. (2016) DiffSharp: An AD library for .NET languages. `arXiv:1611.03423`.

Bendtsen, C. & Stauning, O. (1996) *FADBAD, A Flexible C++ Package for Automatic Differentiation*. Technical Report IMM-REP-1996-17. Department of Mathematical Modelling, Technical University of Denmark, Lyngby, Denmark.

Bischof, C. H., Carle, A., Corliss, G. F., Griewank, A. & Hovland, P. D. (1992) ADIFOR: Generating derivative codes from Fortran programs. *Sci. Program.* **1**(1), 11–29.

Breuleux, O. & van Merriënboer, B. (2017) Automatic differentiation in Myia. In AutoDiff Workshop at Neural Information Processing Systems Conference.

Buckwalter, B. (2007) *Safe Forward-Mode AD in Haskell?* `https://mail.haskell.org/pipermail/haskell-cafe/2007-May/025274.html`.

Chen, T. Q., Rubanova, Y., Bettencourt, J. & Duvenaud, D. (2018) Neural ordinary differential equations. `arXiv:1806.07366`.

Cheney, J. (2012) A dependent nominal type theory. `arXiv:1201.5240`.

Church, A. (1941) *The Calculi of Lambda Conversion*. Princeton, NJ: Princeton University Press.

Clifford, W. K. (1873) Preliminary sketch of bi-quaternions. *Proc. London Math. Soc.* **4**, 381–395.

Ehrhard, T. & Regnier, L. (2003) The differential lambda-calculus. *Theor. Comput. Sci.* **309**(1–3), 1–41.

Elliott, C. M. (2009) Beautiful differentiation. In *International Conference on Functional Programming (ICFP)*. New York, NY, USA: Association for Computing Machinery (ACM).

Elliott, C. M. (2017) Compiling to categories. *International Conference on Functional Programming (ICFP)* New York, NY, USA: Association for Computing Machinery (ACM).

Farr, W. M. (2006) *"Automatic Differentiation" in OCaml*. http://wmfarr.blogspot.com/2006/10/automatic-differentiation-in-ocaml.html.

Griewank, A. & Walther, A. (2008) *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. Philadelphia, PA: Society for Industrial and Applied Mathematics.

Hamilton, W. R. (1837) Theory of conjugate functions, or algebraic couples; with a preliminary and elementary essay on algebra as the science of pure time. *Trans. R. Ir. Acad.* **17**, 293–422.

Hascoët, L. & Pascual, V. (2004) *TAPENADE 2.1 user's guide*. Rapport technique 300. INRIA, Sophia Antipolis.

Karczmarczuk, J. (2001) Functional differentiation of computer programs. *Higher-Order Symbolic Comput.* **14**, 35–57.

Kelly, R., Pearlmutter, B. A. & Siskind, J. M. (2016) Evolving the incremental λ calculus into a model of forward AD. Extended abstract presented at the AD 2016 Conference, Oxford, UK, arXiv:1611.03429.

Kmett, E. (2010) *ad: Automatic Differentiation*. https://hackage.haskell.org/package/ad.

Lavendhomme, R. (1996) *Basic Concepts of Synthetic Differential Geometry*. Kluwer Academic.

Leibniz, G. W. (1684) Nova methodus pro maximis et minimis, itemque tangentibus, quae nec fractas nec irrationales quantitates moratur, et singulare pro illis calculi genus (A new method for maxima and minima, and for tangents, that is not hindered by fractional or irrational quantities, and a singular kind of calculus for the above mentioned). *Acta Eruditorum*.

Maclaurin, D., Duvenaud, D. & Adams, R. P. (2015a) Autograd: Effortless gradients in NumPy. In *Paper presented at International Conference on Machine Learning AutoML Workshop*.

Maclaurin, D., Duvenaud, D. & Adams, R. P. (2015b) Gradient-based hyperparameter optimization through reversible learning. arXiv:1502.03492.

Manzyuk, O. (2012a) A simply typed λ-calculus of forward automatic differentiation. In *Proceedings of the 28th Conference on the Mathematical Foundations of Programming Semantics (MFPS XXVIII)*, Electronic Notes in Theoretical Computer Science, vol. 286, pp. 257–272.

Manzyuk, O. (2012b) Tangent bundles in differential λ-categories. arXiv:1202.0411.

Newton, I. (1704) De quadratura curvarum. In *Opticks: or, A Treatise of the Reflexions, Refractions, Inflexions and Colours of Light, also Two Treatises of the Species and Magnitude of Curvilinear Figures*, London: Printed for Sam Smith and Benjamin Walford, printers to the Royal Society, at the Prince's Arms in St. Paul's Churchyard. Appendix.

Pearlmutter, B. A. & Siskind, J. M. (2007) Lazy multivariate higher-order forward-mode AD. In *Symposium on Principles of Programming Languages*, New York, NY, USA: Association for Computing Machinery (ACM), pp. 155–160.

Pearlmutter, B. A. & Siskind, J. M. (2008) Using programming language theory to make AD sound and efficient. In *International Conference on Automatic Differentiation*, SIAM, pp. 79–90.

Pitts, A. M. (2003) Nominal logic, a first order theory of names and binding. *Inf. Comput.* **186**(2), 165–193.

Plotkin, G. (2018) Some principles of differential programming languages. POPL 2018 Keynote talk, Jan 11, Los Angeles, CA, USA.

Raissi, M. (2018) Deep hidden physics models: Deep learning of nonlinear partial differential equations. *J. Mach. Learn. Res.* **19**(25), 1–24.

Salman, H., Yadollahpour, P., Fletcher, T. & Batmanghelich, K. (2018) Deep diffeomorphic normalizing flows. arXiv:1810.03256.

Shan, C.-c. (2008) *Differentiating Regions*. http://conway.rutgers.edu/~ccshan/wiki/blog/posts/Differentiation/.

Siskind, J. M. & Pearlmutter, B. A. (2005) Perturbation confusion and referential transparency: Correct functional implementation of forward-mode AD. In *Implementation and Application of Functional Languages*, pp. 1–9. Trinity College Dublin Computer Science Department Technical Report TCD-CS-2005-60.

Siskind, J. M. & Pearlmutter, B. A. (2007) First-class nonstandard interpretations by opening closures. In *Symposium on Principles of Programming Languages*, New York, NY, USA: Association for Computing Machinery (ACM), pp. 71–76.

Siskind, J. M. & Pearlmutter, B. A. (2008) Nesting forward-mode AD in a functional framework. *Higher-Order Symbolic Comput.* **21**(4), 361–376.

Speelpenning, B. (1980) *Compiling Fast Partial Derivatives of Functions Given by Algorithms*. PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign.

Sussman, G. J., Abelson, H., Wisdom, J., Katzenelson, J., Mayer, M. E., Hanson, C. P., Halfant, M., Siebert, B., Rozas, G. J., Skordos, P., Koniaris, K., Lin, K. & Zuras, D. (1997a) *Scheme Mechanics Installation for GNU/Linux or Mac OS X*. `http://groups.csail.mit.edu/mac/users/gjs/6946/linux-install.htm`. `http://groups.csail.mit.edu/mac/users/gjs/6946/scmutils-tarballs/`.

Sussman, G. J., Abelson, H., Wisdom, J., Katzenelson, J., Mayer, M. E., Hanson, C. P., Halfant, M., Siebert, B., Rozas, G. J., Skordos, P., Koniaris, K., Lin, K. & Zuras, D. (1997b) *SCMUTILS Reference Manual*. `http://groups.csail.mit.edu/mac/users/gjs/6946/refman.txt`.

Sussman, G. J., Wisdom, J. & Mayer, M. E. (2001) *Structure and Interpretation of Classical Mechanics*. Cambridge, MA: MIT Press.

Sussman, G. J., Wisdom, J. & Farr, W. M. (2013) *Functional Differential Geometry*. Cambridge, MA: MIT Press.

Taylor, B. (1715) *Methodus incrementorum directa et inversa*. London: Typis Pearsonianis.

van Merriënboer, B., Moldovan, D. & Wiltschko, A. (2018) Tangent: Automatic differentiation using source-code transformation for dynamically typed array programming. In *Advances in Neural Information Processing Systems*, Red Hook, New York, USA: Curran Associates, pp. 6259–6268.

Wengert, R. E. (1964) A simple automatic derivative evaluation program. *Commun. ACM* **7**(8), 463–464.

Appendix: Minimal implementation

The repository `https://github.com/qobi/amazing`, file implementation.ss, also available as supplementary material, contains a minimal implementation. It is not intended as a full practical implementation but rather has the expository purpose of explaining the ideas presented in this manuscript. The implementations of `list-real->real` and `list-real*real->real` are similar to those by Siskind & Pearlmutter (2008, Fig. 2). Setting both `*eta-expansion?*` and `*tag-substitution?*` to `#f` uses the implementation of $\mathcal{D}$ in (8d), the implementation of $\overrightarrow{\mathcal{J}}$ in (33), the implementation of **tg** for functions in (8e), and the implementation of **bun** for functions in (30b) and illustrates the bug in (12a)–(12o) and (13). Setting `*eta-expansion?*` to `#t` implements the first solution, eta expansion, from Section 7 and uses the implementation of $\mathcal{D}$ in (18a) and (18b), instead of that in (8d), and the implementation of $\overrightarrow{\mathcal{J}}$ in (32a) and (32b), instead of that in (33). This resolves the bug and yields the correct result (19a)–(19z). Here, $\mathcal{D}$ and $\overrightarrow{\mathcal{J}}$ each use a single side effect to generate ϵs. Instead, setting `*tag-substitution?*` to `#t` implements the second solution, tag substitution, from Section 8 and uses the implementation of **tg** for functions in (23), instead of that in (8e), and the implementation of **bun** for functions in (31), instead of that in (30b). This resolves the bug and yields the correct result (25a)–(25w). Here, $\mathcal{D}$, $\overrightarrow{\mathcal{J}}$, **tg**, **bun**, and tag substitution for functions each use a single side effect to generate ϵs. Setting `*section9?*` to `#t` implements the generalization in Section 9 and uses the implementation of $\mathcal{D}$ in (34) instead of those in (8d) or (18a) and (18b). This works with either solution but exhibits the bug when both solutions are disabled. In all cases, the function whose derivative is taken is pure. This illustrates that the bug can be addressed even when an impure mechanism is used to generate εs. When setting `*tag-substitution?*` to `#t`, setting `*function-substitution*` to `equation-38` uses (38) and gives the wrong result for (39) and (40), setting `*function-substitution*` to `equation-41` uses (41) and illustrates the bug in (42k), while setting `*function-substitution*` to `equation-24d` uses (24d), gives the correct result for (39) and (40), and upholds (42j).