Confusion of Tagged Perturbations in Forward Automatic Differentiation of Higher-Order Functions

Oleksandr Manzyuk, Barak A. Pearlmutter, Alexey Andreyevich Radul and David R. Rush
Hamilton Institute & Department of Computer Science, NUI Maynooth, Co. Kildare, Ireland

Jeffrey Mark Siskind
School of Electrical and Computer Engineering, Purdue University, West Lafayette IN 47907-2035, USA

Abstract. Forward Automatic Differentiation (AD) is a technique for augmenting programs to both perform their original calculation and compute the directional derivative. The essence of Forward AD is to attach a derivative value to each number, and propagate these through the computation. When derivatives are nested, the distinct derivative calculations, and their associated attached values, must be distinguished. In dynamic languages this is typically accomplished by creating a unique tag for each application of the derivative operator, tagging the attached values, and overloading the arithmetic operators. We exhibit a subtle bug, present in fielded implementations, in which perturbations are confused *despite* the tagging machinery.

Keywords: lambda calculus, differential geometry, symbolic computation

1. Forward AD using Tagged Tangents

We would like to add a general first-class Forward Mode Automatic Differentiation operator to an eager dynamic language, for instance, Scheme or Python. Forward AD [10] computes the derivative of a function $f : \mathbb{R} \to \alpha$ at a point c by evaluating $f(c + \varepsilon)$ under a nonstandard interpretation that associates a conceptually infinitesimal perturbation with each real number, propagates these augmented values according to the rules of calculus [3], and extracts the perturbation of the result. When x is a number, we use $x + \bar{x}\varepsilon$ to denote an element of the tangent bundle: the primal value x paired with the tangent value $\bar{x}$, where $\bar{x}$ has the same type as x. We consider such an element of the tangent bundle to also be a number, with arithmetic defined by regarding it as a truncated power series, or equivalently, by taking $\varepsilon^2 = 0$ but $\varepsilon \neq 0$. This implies that $f(x + \bar{x}\varepsilon) = f(x) + \bar{x}f'(x)\varepsilon$ where $f'(x)$ is the first derivative of f at x [6].

© 2015 *Kluwer Academic Publishers. Printed in the Netherlands.*

We can define a first-derivative operator[1] by

$$\mathcal{D}\ f\ x = \mathbf{tg}\ \varepsilon\ (f\ (x + \varepsilon)) \qquad \text{where } \varepsilon \text{ is fresh} \tag{1}$$

with juxtaposition indicating function application, which is left associative. In order for $\mathcal{D}$ to nest correctly we must distinguish between different sets of tangent spaces (or equivalently, distinct differential algebras; or equivalently, distinct generators in a differential algebra) introduced by different invocations of $\mathcal{D}$ [2], which can be implemented by tagging [7, 8]. We will indicate different tags by different subscripts on ε. The tangent extraction function **tg** extracts the tangent part of an element of a tangent bundle, with the appropriate tag indicated in the first argument.

The tangent part of an element of the tangent bundle of a numeric space is:

$$\mathbf{tg}\ \varepsilon\ (a + b\varepsilon) \stackrel{\triangle}{=} b \tag{2}$$

When the primal part of an element of a tangent bundle is a function, this means the tangent bundle is of a function space. In this case elements of the tangent bundle are themselves functions, with **tg** defined by post-composition:

$$\mathbf{tg}\ \varepsilon\ (\lambda x\ .\ e) \stackrel{\triangle}{=} \lambda x\ .\ \mathbf{tg}\ \varepsilon\ e \tag{3}$$

This is the technique used to implement Forward AD in dynamic languages: arithmetic operators are overloaded to handle the chosen representation of elements of tangent bundles over numbers, with tags generated using "`gensym`" or an analogous mechanism.

2. A Bug

If we have properly defined $\mathcal{D}$ and **tg**, then we can reasonably expect to use them to calculate correct derivatives in commonly occurring mathematical situations. In particular, if we define an offset operator:

$$s : \mathbb{R} \to (\mathbb{R} \to \alpha) \to (\mathbb{R} \to \alpha)$$
$$s\ u\ f\ x \stackrel{\triangle}{=} f\ (x + u) \tag{4}$$

[1] The type signature would be $\mathcal{D} : (\mathbb{R} \to \alpha) \to \mathbb{R} \to \alpha'$ where α' is the tangent space of α at a point in α. It is natural to equate $\mathbb{R}' = \mathbb{R}$. Because we only consider $\mathbb{R}$ and functions built on $\mathbb{R}$, we equate $(\alpha \to \beta)' = \alpha \to \beta'$, and Church encoding implies that $(\alpha \times \beta)' = \alpha' \times \beta'$, we can in all present examples equate $\alpha' = \alpha$. A full treatment of this topic is beyond our present scope.

the derivative of s at zero should be the same as the derivative operator: if we define

$$\hat{\mathcal{D}} \triangleq \mathcal{D}\ s\ 0 \tag{5}$$

then $\hat{\mathcal{D}} = \mathcal{D}$ should hold, since

$$\mathcal{D}\ f\ y = \mathbf{tg}\ \varepsilon\ (f\ (y+\varepsilon)) = \mathbf{tg}\ \varepsilon\ (f(y) + f'(y)\varepsilon) = f'(y) \tag{6a}$$

$$\begin{aligned}\hat{\mathcal{D}}\ f\ y = \mathcal{D}\ s\ 0\ f\ y &= \tfrac{\mathrm{d}}{\mathrm{d}u}[\ s\ u\ f\ y\ ]_{u=0} \\ &= \tfrac{\mathrm{d}}{\mathrm{d}u}[\ f(y+u)\ ]_{u=0} = f'(y)\end{aligned} \tag{6b}$$

Unfortunately, as we shall see, the above can exhibit a subtle bug:

$$\hat{\mathcal{D}}\ (\hat{\mathcal{D}}\ f)\ x = 0 \neq \mathcal{D}\ (\mathcal{D}\ f)\ x = f''(x) \tag{7}$$

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.

Note that

$$\begin{aligned}\hat{\mathcal{D}} = \mathcal{D}\ s\ 0 &= \mathbf{tg}\ \varepsilon\ (s\ (0+\varepsilon)) \\ &= \mathbf{tg}\ \varepsilon\ (\lambda f\ .\ \lambda x\ .\ f\ (x+\varepsilon)) \\ &= \lambda f\ .\ \lambda x\ .\ \mathbf{tg}\ \varepsilon\ (f\ (x+\varepsilon))\end{aligned} \tag{8}$$

Assuming that $g : \mathbb{R} \to \mathbb{R}$ we can substitute using (8) and then reduce:

$$\hat{\mathcal{D}}\ (\hat{\mathcal{D}}\ g)\ y = (\lambda f\ .\ \lambda x\ .\ \mathbf{tg}\ \varepsilon\ (f\ (x+\varepsilon))) \ ((\lambda f\ .\ \lambda x\ .\ \mathbf{tg}\ \varepsilon\ (f\ (x+\varepsilon)))\ g)\ y \tag{9a}$$

$$= (\lambda f\ .\ \lambda x\ .\ \mathbf{tg}\ \varepsilon\ (f\ (x+\varepsilon))) \ (\lambda x\ .\ \mathbf{tg}\ \varepsilon\ (g\ (x+\varepsilon)))\ y \tag{9b}$$

$$= (\lambda x\ .\ \mathbf{tg}\ \varepsilon\ ((\lambda x\ .\ \mathbf{tg}\ \varepsilon\ (g\ (x+\varepsilon)))\ (x+\varepsilon)))\ y \tag{9c}$$

$$= \mathbf{tg}\ \varepsilon\ ((\lambda x\ .\ \mathbf{tg}\ \varepsilon\ (g\ (x+\varepsilon)))\ (y+\varepsilon)) \tag{9d}$$

$$= \mathbf{tg}\ \varepsilon\ (\mathbf{tg}\ \varepsilon\ (g\ ((y+\varepsilon)+\varepsilon))) \tag{9e}$$

$$= \mathbf{tg}\ \varepsilon\ (\mathbf{tg}\ \varepsilon\ (g\ (y+2\varepsilon))) \tag{9f}$$

$$= \mathbf{tg}\ \varepsilon\ (\mathbf{tg}\ \varepsilon\ (g(y) + 2g'(y)\varepsilon)) \tag{9g}$$

$$= \mathbf{tg}\ \varepsilon\ (2g'(y)) \tag{9h}$$

$$= 0 \tag{9i}$$

This went wrong, yielding 0 instead of $g''(y)$, because the tag ε was generated exactly *once*, when the definition of $\hat{\mathcal{D}}$ was reduced to normal form in (8). The instantiation of $\mathcal{D}$ 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 (9f). We have two nested applications of **tg** for ε, but for correctness these should be distinctly tagged: ε_1 vs ε_2. If $\hat{\mathcal{D}}$ were not already reduced to normal form, and we instead substitute its definition, then $\mathcal{D}$ will be instantiated twice, giving two fresh tags and a correct result:

$$\begin{aligned}
\hat{\mathcal{D}}\ (\hat{\mathcal{D}}\ g)\ y &= \mathcal{D}\ s\ 0\ (\mathcal{D}\ s\ 0\ g)\ y && \text{(10a)}\\
&= (\lambda f\ .\ \lambda x\ .\ \mathbf{tg}\ \varepsilon_1\ (f\ (x + \varepsilon_1)))\\
&\qquad ((\lambda f\ .\ \lambda x\ .\ \mathbf{tg}\ \varepsilon_2\ (f\ (x + \varepsilon_2)))\ g)\ y && \text{(10b)}\\
&= (\lambda f\ .\ \lambda x\ .\ \mathbf{tg}\ \varepsilon_1\ (f\ (x + \varepsilon_1)))\\
&\qquad (\lambda x\ .\ \mathbf{tg}\ \varepsilon_2\ (g\ (x + \varepsilon_2)))\ y && \text{(10c)}\\
&= (\lambda x\ .\ \mathbf{tg}\ \varepsilon_1\ ((\lambda x\ .\ \mathbf{tg}\ \varepsilon_2\ (g\ (x + \varepsilon_2)))\ (x + \varepsilon_1)))\ y && \text{(10d)}\\
&= \mathbf{tg}\ \varepsilon_1\ ((\lambda x\ .\ \mathbf{tg}\ \varepsilon_2\ (g\ (x + \varepsilon_2)))\ (y + \varepsilon_1)) && \text{(10e)}\\
&= \mathbf{tg}\ \varepsilon_1\ (\mathbf{tg}\ \varepsilon_2\ (g\ ((y + \varepsilon_1) + \varepsilon_2))) && \text{(10f)}\\
&= \mathbf{tg}\ \varepsilon_1\ (\mathbf{tg}\ \varepsilon_2\ (g(y + \varepsilon_1) + g'(y + \varepsilon_1)\varepsilon_2)) && \text{(10g)}\\
&= \mathbf{tg}\ \varepsilon_1\ g'(y + \varepsilon_1) && \text{(10h)}\\
&= \mathbf{tg}\ \varepsilon_1\ (g'(y) + g''(y)\varepsilon_1) && \text{(10i)}\\
&= g''(y) && \text{(10j)}
\end{aligned}$$

3. Discussion

In a Forward AD system which uses tags to distinguish instances of $\mathcal{D}$, eta reduction is unsound. The definition $\hat{\mathcal{D}}\ f\ y = \mathcal{D}\ s\ 0\ f\ y$ must not be eta reduced to $\hat{\mathcal{D}} = \mathcal{D}\ s\ 0$, and one must not memoize or hoist $\mathcal{D}\ s\ 0$, as it is impure due to the requirement for a fresh tag. Even the above constraint can be insufficient when $\hat{\mathcal{D}}$ is applied to a function that is not $\mathbb{R} \to \mathbb{R}$ but instead $\mathbb{R} \to \alpha$ for some other α. In fact, expanded variants of $\hat{\mathcal{D}}$ are needed for various α. For instance, applying $\hat{\mathcal{D}}$ to a function $\underbrace{\mathbb{R} \to \cdots \to \mathbb{R}}_{n} \to \mathbb{R}$ requires an eta-reduction-protected

$$\hat{\mathcal{D}}_n\ f\ y_1\ \ldots\ y_n \stackrel{\triangle}{=} \mathcal{D}\ s\ 0\ f\ y_1\ \ldots\ y_n \qquad (11)$$

In general, $\mathcal{D}$ should only be instantiated in a context that contains all arguments necessary to subsequently allow the post-composition of the **tg** introduced by the instantiation of $\mathcal{D}$ to immediately beta reduce to a non-function-containing value. Note that **tg** distributes over aggregates

like tuples and lists, further complicating the determination of when $\mathcal{D}$ can be instantiated.

Another alternative would be to guard the returned function object against tag collision. In a programming language with opaque closures, post-composition must be implemented using a wrapper:

$$\textbf{tg}\ \varepsilon\ (\lambda x\ .\ e) \stackrel{\triangle}{=} \lambda y\ .\ \textbf{tg}\ \varepsilon\ ((\lambda x\ .\ e)\ y) \tag{12}$$

This wrapper can be augmented to guard against the problem we have encountered:

$$\textbf{tg}\ \varepsilon_1\ (\lambda x\ .\ e) \stackrel{\triangle}{=} \lambda y\ .\ (\textbf{swiz}\ \varepsilon_2\ \varepsilon_1\ (\textbf{tg}\ \varepsilon_1\ ((\lambda x\ .\ e)\ (\textbf{swiz}\ \varepsilon_1\ \varepsilon_2\ y)))) \quad \text{where } \varepsilon_2 \text{ is fresh} \tag{13}$$

Here "**swiz** $\varepsilon_1\ \varepsilon_2\ v$" substitutes ε_2 for every occurrence of ε_1 in v. In a language with opaque closures, **swiz** must operate on function objects by appropriate pre- and post-composition. This technique was used to address the present issue in the 30-Aug-2011 release of scmutils, a software package that accompanies a textbook on classical mechanics [9], in response to an early version of this manuscript. Unfortunately the computational burden of such "swizzling" violates the complexity guarantees of Forward AD. This leaves us in the awkward position of there being *no known technique* for implementing Forward AD, with its defining complexity guarantee, and generalized to functions with higher-order outputs (including even curried functions), in a dynamic language.

We have used fresh tags to create a fresh set of tangent spaces each time $\mathcal{D}$ is instantiated. Forward AD implementations in dynamically typed languages which support operator overloading (e.g., Scheme, Python) are susceptible to the problem we have exhibited due to the impurity of "`gensym`." It seems reasonable to speculate that features of advanced static type systems (such as existential types) may be used to prevent this error. However, (a) current static type systems prevent first-class automatic differentiation operators themselves from being defined, and (b) an intuition is a far cry from a proof. It is a current topic of research to satisfactorily define a λ-calculus based system which correctly models Forward AD [1, 4, 5].

Acknowledgements

We thank the anonymous reviewers for helpful feedback and suggestions. This work was supported, in part, by Science Foundation Ireland

Principal Investigator grant 09/IN.1/I2637 and Army Research Laboratory Cooperative Agreement Number W911NF-10-2-0060. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either express or implied, of SFI, ARL, or the Irish or U.S. Governments. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes, notwithstanding any copyright notation herein.

References

1. Ehrhard, T. and L. Regnier: 2003, 'The Differential Lambda-Calculus'. *Theoretical Computer Science* **309**(1-3), 1–41.
2. Lavendhomme, R.: 1996, *Basic Concepts of Synthetic Differential Geometry*. Kluwer Academic.
3. Leibniz, G. W.: 1664, 'A new method for maxima and minima as well as tangents, which is impeded neither by fractional nor irrational quantities, and a remarkable type of calculus for this'. *Acta Eruditorum*.
4. Manzyuk, O.: 2012a, 'A Simply Typed λ-Calculus of Forward Automatic Differentiation'. In: *Mathematical Foundations of Programming Semantics Twenty-eighth Annual Conference*. Bath, UK, pp. 259–73.
5. Manzyuk, O.: 2012b, 'Tangent bundles in differential λ-categories'. Technical Report arXiv:1202.0411.
6. Newton, I.: 1704, 'De quadratura curvarum'. In *Optiks*, 1704 edition. Appendix.
7. Siskind, J. M. and B. A. Pearlmutter: 2005, 'Perturbation Confusion and Referential Transparency: Correct Functional Implementation of Forward-Mode AD'. In: A. Butterfield (ed.): *Implementation and Application of Functional Languages—17th International Workshop, IFL'05*. Dublin, Ireland, pp. 1–9. Trinity College Dublin Computer Science Department Technical Report TCD-CS-2005-60.
8. Siskind, J. M. and B. A. Pearlmutter: 2008, 'Nesting Forward-Mode AD in a Functional Framework'. *Higher-Order and Symbolic Computation* **21**(4), 361–76.
9. Sussman, G. J., J. Wisdom, and M. E. Mayer: 2001, *Structure and Interpretation of Classical Mechanics*. Cambridge, MA: MIT Press.
10. Wengert, R. E.: 1964, 'A simple automatic derivative evaluation program'. *Comm. of the ACM* **7**(8), 463–4.