Perturbation Confusion and Referential Transparency

Correct Functional Implementation of Forward-Mode AD

Jeffrey Mark Siskind[1] and Barak A. Pearlmutter[2]

[1] School of Electrical and Computer Engineering, Purdue University, USA
[2] Hamilton Institute, NUI Maynooth, Ireland
[1] qobi@purdue.edu [2] barak@cs.nuim.ie

Abstract. It is tempting to incorporate differentiation operators into functional-programming languages. Making them first-class citizens, however, is an enterprise fraught with danger. We discuss a potential problem with forward-mode AD common to many AD systems, including all attempts to integrate a forward-mode AD operator into HASKELL. In particular, we show how these implementations fail to preserve referential transparency, and can compute grossly incorrect results when the differentiation operator is applied to a function that itself uses that operator. The underlying cause of this problem is *perturbation confusion*, a failure to distinguish between distinct perturbations introduced by distinct invocations of the differentiation operator. We then discuss how perturbation confusion can be avoided.

1 Introduction

Referential transparency is the battle cry of the non-strict functional programming community. A subtle issue with referential transparency can arise when a derivative-taking operator is combined with functional programming. As an illustration of the issue consider the following expression, whose value should obviously be 1.

$$\left.\frac{\mathrm{d}}{\mathrm{d}x}\left(x\left(\left.\frac{\mathrm{d}}{\mathrm{d}y}\,x+y\right|_{y=1}\right)\right)\right|_{x=1} \stackrel{?}{=} 2 \tag{1}$$

This would be expressed in a functional-programming language as

$$\mathcal{D}\;(\lambda x\;.\;x \times (\mathcal{D}\;(\lambda y\;.\;x+y)\;1))\;1 \tag{2}$$

where $\times$ indicates multiplication and $\mathcal{D}$ is a derivative-taking operator.

$$\mathcal{D}\;f\;c = \left.\frac{\mathrm{d}}{\mathrm{d}x}\,f(x)\right|_{x=c} \tag{3}$$

Automatic Differentiation (AD) is an established enterprise for calculating derivatives of functions expressed as computer programs (Griewank 2000).

Forward-mode AD (Wengert 1964) implements $\mathcal{D}$ by evaluating $f\ (c+\varepsilon)$ under an abstract interpretation that associates a conceptually infinitesimal perturbation with each real number, propagates them according to the rules of calculus (Leibniz 1664; Newton 1704), and extracts the perturbation of the result.

To see how this works, let us manually apply the mechanism to a simple expression. We use $x + x'\varepsilon$ to denote x with associated perturbation x', by analogy with the standard $a + b\mathrm{i}$ for complex numbers.

$$\begin{aligned}
\left.\frac{\mathrm{d}}{\mathrm{d}x}\ x^2+x+1\right|_{x=3} &= \mathcal{D}\ (\lambda x\ .\ x \times x + x + 1)\ 3 \\
&= \mathcal{E}\ ((\lambda x\ .\ x \times x + x + 1)\ (3+\varepsilon)) \\
&= \mathcal{E}\ ((3+\varepsilon) \times (3+\varepsilon) + (3+\varepsilon) + 1) \\
&= \mathcal{E}\ ((9+6\varepsilon) + (3+\varepsilon) + 1) \\
&= \mathcal{E}\ (13+7\varepsilon) \\
&= 7
\end{aligned}$$

where $\mathcal{E}\ (x + x'\varepsilon) \stackrel{\triangle}{=} x'$ and $\mathcal{D}\ f\ c \stackrel{\triangle}{=} \mathcal{E}\ (f\ (c+\varepsilon))$. This is the essence of forward-mode AD.

2 Perturbation Confusion

We can now evaluate (2) using this machinery.

$$\begin{aligned}
&\mathcal{D}\ (\lambda x\ .\ x \times (\mathcal{D}\ (\lambda y\ .\ x + y)\ 1))\ 1 \\
&\quad = \mathcal{E}\ ((\lambda x\ .\ x \times (\mathcal{D}\ (\lambda y\ .\ x + y)\ 1))\ (1+\varepsilon)) && (4) \\
&\quad = \mathcal{E}\ ((1+\varepsilon) \times (\mathcal{D}\ (\lambda y\ .\ (1+\varepsilon) + y)\ 1)) && (5) \\
&\quad = \mathcal{E}\ ((1+\varepsilon) \times (\mathcal{E}\ ((\lambda y\ .\ (1+\varepsilon) + y)\ (1+\varepsilon)))) && (6) \\
&\quad = \mathcal{E}\ ((1+\varepsilon) \times (\mathcal{E}\ ((1+\varepsilon) + (1+\varepsilon)))) && (7) \\
&\quad = \mathcal{E}\ ((1+\varepsilon) \times (\mathcal{E}\ (2+2\varepsilon))) && (8) \\
&\quad = \mathcal{E}\ ((1+\varepsilon) \times 2) && (9) \\
&\quad = \mathcal{E}\ (2+2\varepsilon) && (10) \\
&\quad = 2 && (11) \\
&\quad \neq 1
\end{aligned}$$

The technique described by Karczmarczuk (1998, 2001) and adopted by Nilsson (2003) exhibits this problem; see Appendix A. The underlying issue is *perturbation confusion*, a failure (at step 8) to distinguish between distinct perturbations introduced by distinct invocations of $\mathcal{D}$.

3 Tagging Avoids Perturbation Confusion

One way to remedy perturbation confusion is to define

$$\mathcal{D}\ f\ c \stackrel{\triangle}{=} \mathcal{E}_t\ (f\ (c+\varepsilon_t)) \qquad (12)$$

where t is a tag unique to each invocation of $\mathcal{D}$, and define

$$\mathcal{E}_t\ (x + x'\varepsilon_t) \stackrel{\triangle}{=} x' \tag{13}$$

to extract only the correspondingly tagged perturbation, ignoring any others; see Appendix C. We can evaluate (2) using these tags.

$$\begin{aligned}
&\mathcal{D}\ (\lambda x\ .\ x \times (\mathcal{D}\ (\lambda y\ .\ x + y)\ 1))\ 1\\
&\quad = \mathcal{E}_a\ ((\lambda x\ .\ x \times (\mathcal{D}\ (\lambda y\ .\ x + y)\ 1))\ (1 + \varepsilon_a)) && (14)\\
&\quad = \mathcal{E}_a\ ((1 + \varepsilon_a) \times (\mathcal{D}\ (\lambda y\ .\ (1 + \varepsilon_a) + y)\ 1)) && (15)\\
&\quad = \mathcal{E}_a\ ((1 + \varepsilon_a) \times (\mathcal{E}_b\ ((\lambda y\ .\ (1 + \varepsilon_a) + y)\ (1 + \varepsilon_b)))) && (16)\\
&\quad = \mathcal{E}_a\ ((1 + \varepsilon_a) \times (\mathcal{E}_b\ ((1 + \varepsilon_a) + (1 + \varepsilon_b)))) && (17)\\
&\quad = \mathcal{E}_a\ ((1 + \varepsilon_a) \times (\mathcal{E}_b\ (2 + \varepsilon_a + \varepsilon_b))) && (18)\\
&\quad = \mathcal{E}_a\ ((1 + \varepsilon_a) \times 1) && (19)\\
&\quad = \mathcal{E}_a\ (1 + \varepsilon_a) && (20)\\
&\quad = 1 && (21)
\end{aligned}$$

Note how the erroneous addition of distinct perturbations (step 8) is circumvented at the corresponding point here (step 18).

4 Referential Transparency

Perturbation confusion can violate referential transparency. Consider

$$c\ x \stackrel{\triangle}{=} \mathcal{D}\ (\lambda y\ .\ x + y)\ 1$$

which should have a constant value of 1 regardless of its numeric argument x. Therefore $\lambda x\ .\ x \times (c\ x)$ and $\lambda x\ .\ x \times 1$ should both denote the identity function for numbers. However, as seen above and in Appendix A,

$$\mathcal{D}\ (\lambda x\ .\ x \times (c\ x))\ 1$$

and

$$\mathcal{D}\ (\lambda x\ .\ x \times 1)\ 1$$

yield different results when distinct perturbations are not distinguished.

5 Related Work

Forward-mode AD was implemented in Scheme as part of the scmutils package included in the instructional materials associated with a textbook on classical mechanics (Sussman et al. 2001). Scmutils is neither documented nor published, but examination of its uncommented source code reveals an explicit tagging mechanism to distinguish distinct perturbations, and scmutils correctly evaluates (2).

Explicit tagging of the sort described above is impossible to implement in a purely functional language. Such explicit tagging, however, is not necessary to remedy perturbation confusion. A broad class of implemented forward-mode AD systems operate by performing a *static* abstract interpretation of the original program, to pair perturbations with real values, via a source-to-source transformation, overloading, or some combination of the two. Source-to-source transformation can be performed *inter alia* by a preprocessor, as in ADIFOR (Bischof et al. 1996), ADIC (Bischof et al. 1997), and ADiMat (Bischof et al. 2003), or by an ad-hoc reflective mechanism in an underlying interpreter, as in GRADIENT (Monagan and Neuenschwander 1993).

6 Static Avoidance of Perturbation Confusion

Static abstract interpretation using a source-to-source transformation can remedy perturbation confusion in a functional framework. The general idea is to wrap n calls to `lift` around each numeric variable reference made inside the function passed to the `d` operator, where n is the number of calls to `d` that intervene between that variable's definition and its reference. Doing this transforms

```
constant_one x = d (\y -> x + y) 1

d (\x -> x * (constant_one x)) 1
```

into

```
constant_one x = d (\y -> (lift x) + y) 1

d (\x -> x * (constant_one x)) 1
```

which yields the correct result; see Appendix D. This cannot be done automatically in Haskell, but requires a preprocessor. In general, determining the value of each such n requires sophisticated non-local analysis, unless the program is written in an extremely restrictive style to make each such n apparent. Further complications arise when attempting to lift aggregate data objects that contain numeric values and when different control-flow paths to a variable reference can lead to different values of n.

An implementation of AD in a functional framework which incorporates both forward-mode and reverse-mode and supports arbitrary nested application of AD operators has been developed by the authors. Instead of tagging, source-to-source transformations are used to avoid perturbation confusion. The system performs these transformations using a simple reflective API. (2) can be coded directly and transparently in this system as

```
(define (derivative f x) (tangent ((j* f) (bundle x 1))))

(derivative (lambda (x) (* x (derivative (lambda (y) (+ x y)) 1))) 1)
```

which yields the correct result. A detailed discussion of the foundations of this system will be the subject of a forthcoming publication, but the implementation is currently available at `http://www-bcl.cs.nuim.ie/~qobi/stalingrad/`.

Acknowledgements

The authors gratefully acknowledge the support of NSF grant CCF-0438806, Science Foundation Ireland grant 00/PI.1/C067, and the Higher Education Authority of Ireland.

References

Bischof, C., Lang, B., and Vehreschild, A. (2003). Automatic Differentiation for MATLAB Programs. *Proc. Appl. Math. Mech.*, 2(1):50–3.

Bischof, C. H., Carle, A., Khademi, P., and Mauer, A. (1996). ADIFOR 2.0: Automatic differentiation of Fortran 77 programs. *IEEE Computational Science & Engineering*, 3(3):18–32.

Bischof, C. H., Roh, L., and Mauer, A. (1997). ADIC—An extensible automatic differentiation tool for ANSI-C. *Software Practice & Exper.*, 27(12):1427–56.

Griewank, A. (2000). *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation.* Number 19 in Frontiers in Applied Mathematics. SIAM.

Karczmarczuk, J. (1998). Functional differentiation of computer programs. In *Proceedings of the III ACM SIGPLAN International Conference on Functional Programming*, pages 195–203, Baltimore, MD.

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

Leibnitz, G. W. (1664). A new method for maxima and minima as well as tangents, which is impeded neither by fractional nor irrational quantitites, and a remarkable type of calculus for this. *Acta Eruditorum.*

Monagan, M. B. and Neuenschwander, W. M. (1993). GRADIENT: Algorithmic differentiation in Maple. In *International Symposium on Symbolic and Algebraic Computation.*

Newton, I. (1704). De quadrutura curvarum. In *Optiks*, 1704 edition. Appendix.

Nilsson, H. (2003). Functional automatic differentiation with Dirac impulses. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming*, pages 153–64, Uppsala, Sweden. ACM Press.

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

Wengert, R. E. (1964). A simple automatic derivative evaluation program. *Communications of the ACM*, 7(8):463–4.

Appendices

Unabbreviated versions of the below code are available at `http://www-bcl.cs.nuim.ie/~qobi/stalingrad/software/hosc2005/`.

A Incorrect Implementation in Haskell

One would naturally want to write

```
constant_one x = d (\y -> x + y) 1

d (\x -> x * (constant_one x)) 1
```

However, the above yields a type violation at the expression

```
x * (constant_one x)
```

This is because Haskell only automatically coerces constants, not the results of other kinds of expressions. Such automatic coercion can be manually simulated by inserting an appropriate coercion operator at the point of type violation.

```
constant_one x = d (\y -> x + y) 1

d (\x -> x * (lift (constant_one x)) ) 1
```

Note however that while this is now type correct, it yields an incorrect result. A system that automatically introduced such coercions would also necessarily suffer from such perturbation confusion.

```
data Num a => Bundle a = Bundle a a

instance Num a => Show (Bundle a) where
    showsPrec p (Bundle x x') = showsPrec p [x,x']

instance Num a => Eq (Bundle a) where
    (Bundle x x') == (Bundle y y') = (x == y)

lift z = Bundle z 0

instance Num a => Num (Bundle a) where
    (Bundle x x') + (Bundle y y') = Bundle (x + y) (x' + y')
    (Bundle x x') * (Bundle y y') = Bundle (x * y) (x * y' + x' * y)
    fromInteger z = lift (fromInteger z)

instance Fractional a => Fractional (Bundle a) where
    fromRational z = lift (fromRational z)

d f x = let (Bundle y y') = f (Bundle x 1) in y'

constant_one x = d (\y -> x + y) 1

should_be_one_a = d (\x -> x * (lift (constant_one x))) 1
should_be_one_b = d (\x -> x * (lift 1              )) 1

violation_of_referential_transparency = should_be_one_a /= should_be_one_b
```

B Similar Incorrect Implementation in Scheme

The same method can be implemented in Scheme, where it exhibits the same problem.

```
(define primal cadr)

(define tangent caddr)

(define (bundle primal tangent) (list 'bundle primal tangent))

(define bundle?
 (let ((pair? pair?)) (lambda (x) (and (pair? x) (eq? (car x) 'bundle)))))

(set! pair? (lambda (x) (and (pair? x) (not (bundle? x)))))

(define (lift-real x) (bundle x 0))

(define (lift-real->real f df/dx)
 (letrec ((self (lambda (x)
                 (if (bundle? x)
                     (bundle (self (primal x))
                             (* (df/dx (primal x)) (tangent x)))
                     (f x)))))
  self))

(define (lift-real*real->real f df/dx1 df/dx2)
 (letrec ((self
           (lambda (x1 x2)
            (if (bundle? x1)
                (if (bundle? x2)
                    (bundle
                     (self (primal x1) (primal x2))
                     (+ (* (df/dx1 (primal x1) (primal x2))
                           (tangent x1))
                        (* (df/dx2 (primal x1) (primal x2))
                           (tangent x2))))
                    (self x1 (lift-real x2)))
                (if (bundle? x2)
                    (self (lift-real x1) x2)
                    (f x1 x2))))))
  self))

(define (lift-real->boolean f)
 (letrec ((self (lambda (x) (if (bundle? x) (self (primal x)) (f x)))))
  self))

(define (lift-real*real->boolean f)
 (letrec ((self (lambda (x1 x2)
                 (if (bundle? x1)
                     (if (bundle? x2)
                         (self (primal x1) (primal x2))
                         (self (primal x1) x2))
                     (if (bundle? x2) (self x1 (primal x2)) (f x1 x2))))))
  self))

;;; Overloads not needed for this example are omitted.
(set! + (lift-real*real->real + (lambda (x1 x2) 1) (lambda (x1 x2) 1)))
(set! * (lift-real*real->real * (lambda (x1 x2) x2) (lambda (x1 x2) x1)))

(define (derivative f)
 (lambda (x) (let ((y (f (bundle x 1)))) (if (bundle? y) (tangent y) 0))))

(define should-be-one
 ((derivative (lambda (x) (* x ((derivative (lambda (y) (+ x y))) 1)))) 1))
```

C Corrected Implementation in Scheme

Definitions of `primal`, `tangent`, `bundle?`, `pair?`, `lift-real->boolean`, and `lift-real*real->boolean`, the overloads, and `should-be-one` are unchanged from Appendix B. Boxes indicate additions and modifications.

```
(define tag cadddr)

(define (bundle tag primal tangent) (list 'bundle primal tangent tag))

(define make-tag (let ((tag 0)) (lambda () (set! tag (+ tag 1)) tag)))

(define (lift-real tag x) (bundle tag x 0))

(define (in? t x) (and (bundle? x) (or (= (tag x) t) (in? t (primal x)))))

(define (lift-real->real f df/dx)
 (letrec ((self (lambda (x)
                (if (bundle? x)
                    (bundle (tag x)
                            (self (primal x))
                            (* (df/dx (primal x)) (tangent x)))
                    (f x)))))
  self))

(define (lift-real*real->real f df/dx1 df/dx2)
 (letrec ((self
           (lambda (x1 x2)
            (if (bundle? x1)
                (if (bundle? x2)
                    (if (= (tag x1) (tag x2))
                        (bundle
                         (tag x1)
                         (self (primal x1) (primal x2))
                         (+ (* (df/dx1 (primal x1) (primal x2))
                               (tangent x1))
                            (* (df/dx2 (primal x1) (primal x2))
                               (tangent x2))))
                     (if (in? (tag x1) x2)
                         (self (lift-real (tag x2) x1) x2)
                         (self x1 (lift-real (tag x1) x2))))
                    (self x1 (lift-real (tag x1) x2)))
                (if (bundle? x2)
                    (self (lift-real (tag x2) x1) x2)
                    (f x1 x2))))))
  self))

(define (e-t t x)
 (if (bundle? x) (if (= (tag x) t) (tangent x) (e-t t (primal x))) 0))
(define (derivative f)
 (lambda (x) (let ((t (make-tag))) (e-t t (f (bundle t x 1))))))
```

D Corrected Implementation in Haskell

It is possible to correct the problem by manually inserting a coercion operation (`lift`). A method for determining where these are needed is discussed in Section 6. This method only applies to code written so as to maintain a number of very restrictive static properties.

```
data Num a => Bundle a = Bundle a a

instance Num a => Show (Bundle a) where
    showsPrec p (Bundle x x') = showsPrec p [x,x']

instance Num a => Eq (Bundle a) where
    (Bundle x x') == (Bundle y y') = (x == y)

lift z = Bundle z 0

instance Num a => Num (Bundle a) where
    (Bundle x x') + (Bundle y y') = Bundle (x + y) (x' + y')
    (Bundle x x') * (Bundle y y') = Bundle (x * y) (x * y' + x' * y)
    fromInteger z = lift (fromInteger z)

instance Fractional a => Fractional (Bundle a) where
    fromRational z = lift (fromRational z)

d f x = let (Bundle y y') = f (Bundle x 1) in y'

constant_one x = d (\y -> (lift x) + y) 1

should_be_one_a = d (\x -> x * (constant_one x)) 1
should_be_one_b = d (\x -> x * 1               ) 1

violation_of_referential_transparency = should_be_one_a /= should_be_one_b
```