Higher Order and Symbolic Computation manuscript No.
(will be inserted by the editor)

Nesting Forward-Mode AD in a Functional Framework

Jeffrey Mark Siskind · **Barak A. Pearlmutter**

Received: 18 August 2005 / Accepted: 16 April 2007

Abstract We discuss the augmentation of a functional-programming language with a derivative-taking operator implemented with forward-mode automatic differentiation (AD). The primary technical difficulty in doing so lies in ensuring correctness in the face of nested invocation of that operator, due to the need to distinguish perturbations introduced by distinct invocations. We exhibit a series of implementations of a referentially-transparent forward-mode-AD derivative-taking operator, each of which uses a different non-referentially-transparent mechanism to distinguish perturbations. Even though the forward-mode-AD derivative-taking operator is itself referentially transparent, we hypothesize that one cannot correctly formulate this operator as a function definition in current pure dialects of HASKELL.

Keywords Automatic differentiation, Applicative (functional) languages, Referential transparency, Multiple transformation

Mathematics Subject Classification (2000) 68N18 · 26A24 · 68W30

Supported, in part, by US NSF grant CCF-0438806, Science Foundation Ireland grant 00/PI.1/C067, and the Higher Education Authority of Ireland (An tÚdarás Um Ard-Oideachas). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the supporting agencies.

J. M. Siskind
School of Electrical and Computer Engineering, Purdue University, USA
Tel.: +1 765 496 3197
Fax: +1 765 494 6440
E-mail: qobi@purdue.edu

B. A. Pearlmutter
Hamilton Institute, NUI Maynooth, Ireland
Tel.: +353 1 708 6100
Fax: +353 1 708 6269
E-mail: barak@cs.nuim.ie

1 Introduction

The ability to nest function invocation is central to functional programming. One would be discontent with a language or implementation that would not allow one to use a nested invocation of MAP to compute outer products.

$$\textsc{OuterProduct}\ f\ \mathbf{x}\ \mathbf{y} \stackrel{\triangle}{=} \textsc{Map}\ (\lambda x\ .\ \textsc{Map}\ (\lambda y\ .\ f\ x\ y)\ \mathbf{y})\ \mathbf{x}$$

In an analogous fashion, one would expect to be able to write

$$\textsc{Min}\ (\lambda x\ .\ (f\ x) + \textsc{Min}\ (\lambda y\ .\ g\ x\ y)) \tag{1}$$

given a definition for MIN that takes a suitable function $\mathbb{R} \rightarrow \mathbb{R}$ as its argument and returns (an approximation to) a (local) minimum of that function. Correct processing of either of the above requires correct handling of nested function invocation. In particular, the outer call to MAP or MIN is passed an outer function that itself calls MAP or MIN on an inner function that depends on the argument x of the outer function.

Suppose our implementation of MIN uses gradient descent. It would be desirable for MIN, which takes f as a functional argument, to be able to use the derivative of f without the caller's knowledge. Thus, it would be advantageous for a system to provide a higher-order function $\mathcal{D}$ that maps functions to their derivatives. With such a facility, (1) would take the form

$$\ldots \mathcal{D}\ (\lambda x\ .\ \ldots\ \mathcal{D}\ (\lambda y\ .\ g\ x\ y) \ldots) \ldots$$

This requires that nested invocation of $\mathcal{D}$ operate correctly.

Automatic Differentiation (AD), and in particular forward-mode AD (Wengert, 1964), is one established method for computing derivatives and can be used to implement $\mathcal{D}$. The remainder of this paper discusses issues surrounding such an implementation, and uses $\mathcal{D}$ to refer to the notion of a derivative-taking operator implemented using forward-mode AD. We hypothesize that it is not possible to formulate a $\mathcal{D}$ that properly nests as a function definition in current pure dialects of HASKELL. This is somewhat ironic, as while $\mathcal{D}$ can be implemented using one of several alternate non-referentially-transparent mechanisms, $\mathcal{D}$ itself *is* referentially transparent.[1]

The remainder of this paper elaborates on the above observations. We begin with a brief overview of forward-mode AD in section 2. We then show how to implement $\mathcal{D}$ as a procedure definition in SCHEME, in a way that can properly nest. To do this we first construct an API to the necessary data structures, in section 3, and then use this machinery to build a forward-mode AD engine and drive it using standard SCHEME procedure names via overloading, in section 4. This implementation uses only one non-referentially-transparent side effect. We discuss, in section 5, a number of alternate non-referentially-transparent mechanisms that suffice to implement $\mathcal{D}$. It is noted in

[1] There are subtle differences between $\mathcal{D}$ and the classical derivative-taking operator in mathematics. For example, given the definition $f\ x \stackrel{\triangle}{=} \textbf{if}\ x = c\ \textbf{then}\ c\ \textbf{else}\ x$ the derivative of f at c is 1, yet $\mathcal{D}\ f\ c = 0$. Like all mathematical notions, classical differentiation is referentially transparent, since the derivative of a function is defined on its extension rather than its intension. Furthermore, $\mathcal{D}$ is also referentially transparent in the sense that if t_1 and t_2 are semantically equivalent, then $\mathcal{D}\ t_1$ and $\mathcal{D}\ t_2$ are also semantically equivalent. (Note that the presence of the $=$ predicate in the antecedent of the conditional in the definition of f does not license β-substitution, because that predicate does not necessarily correspond to semantic equivalence.)

section 6 that, in certain cases, static analysis or program transformation can allow nested invocation of $\mathcal{D}$ without non-referentially-transparent mechanisms. We give an example that utilizes nested invocation of $\mathcal{D}$ in section 7. We conclude, in section 8, with a discussion of the history and implications of the desire to incorporate differentiation into functional programming.

2 Forward-Mode AD as Nonstandard Interpretation

Forward-mode AD computes the derivative of a function f 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 (Leibnitz, 1664; Newton, 1704), and extracts the perturbation of the result. We use $x+x'\varepsilon$ to denote a *dual number* (Clifford, 1873), i.e. x with associated perturbation x', by analogy with the standard notation $a+b\mathrm{i}$ for complex numbers.[2] To see how this works, let us manually apply the mechanism to a simple expression.

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

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.[3]

[2] Just as arithmetic on complex numbers $a+b\mathrm{i}$ can be defined by taking $\mathrm{i}^2=-1$, arithmetic on dual numbers $x+x'\varepsilon$ can be defined by taking $\varepsilon^2=0$ but $\varepsilon\neq 0$. Implementations of complex arithmetic typically represent complex numbers $a+b\mathrm{i}$ as Argand pairs $\langle a,b\rangle$, and similarly implementations of forward-mode AD typically represent dual numbers $x+x'\varepsilon$ as tangent-bundle pairs $\langle x,x'\rangle$. Furthermore, just as implementations of complex arithmetic typically overload the arithmetic primitives to manipulate complex numbers, implementations of forward-mode AD typically overload the arithmetic primitives to manipulate dual numbers. One important difference between complex numbers and dual numbers is that while complex numbers can only have real components, as used here components of members of a new dual-number type can be either reals or members of an existing dual-number type.

[3] For expository simplicity, we limit our discussion of forward-mode AD to a special case, namely first derivatives of univariate functions $\mathbb{R}\rightarrow\mathbb{R}$. However, forward-mode immediately generalizes in two different ways. First, vector functions can be handled with the same efficiency and mechanisms as scalar functions by adopting a directional derivative operator, which finds the directional derivative $y':\mathbb{R}^m$ of $f:\mathbb{R}^n\rightarrow\mathbb{R}^m$ at $x:\mathbb{R}^n$ in the direction $x':\mathbb{R}^n$ by calculating $(y_1+y'_1\varepsilon,\ldots,y_m+y'_m\varepsilon)=f\ (x_1+x'_1\varepsilon,\ldots,x_n+x'_n\varepsilon)$ using the same nonstandard interpretation of f on dual numbers as in the scalar case. Second, a dual number can be viewed as a power series that has been truncated at ε^2. One can extend the notion of dual numbers to allow higher-order terms, either by truncating at a higher order or by representing the coefficients of an infinite power series as a stream (Karczmarczuk, 1998a,b, 1999, 2001; Nilsson, 2003; Pearlmutter and Siskind, 2007), thus computing higher-order derivatives. Nested invocation of a first-order derivative-taking operator can also compute higher-order derivatives. However, nested invocation of a first-order derivative-taking operator can compute things that a single invocation of a higher-order derivative-taking operator cannot.

In order for this mechanism to correctly handle nesting, we must distinguish between different perturbations introduced by different invocations of $\mathcal{D}$. One way to do this is to create a hierarchy of dual-number types, distinguished by a distinct ε for each distinct invocation of $\mathcal{D}$. The components of a dual-number type created for a non-nested invocation of $\mathcal{D}$ are reals, while the components of a dual-number type created for a nested invocation of $\mathcal{D}$ are members of the dual-number type of the immediately surrounding invocation of $\mathcal{D}$.

The intuition behind the necessity and sufficiency of such an extension is illustrated by the following example.

$$\begin{aligned}
&\frac{d}{dx}\left(x\left(\frac{d}{dy}\,xy\Big|_{y=2}\right)\right)\Bigg|_{x=1}\\
&\quad= \mathcal{D}\ (\lambda x\ .\ x \times (\mathcal{D}\ (\lambda y\ .\ x \times y)\ 2))\ 1\\
&\quad= \mathcal{E}\ \varepsilon_a\ ((\lambda x\ .\ x \times (\mathcal{D}\ (\lambda y\ .\ x \times y)\ 2))\ (1+\varepsilon_a))\\
&\quad= \mathcal{E}\ \varepsilon_a\ ((1+\varepsilon_a) \times (\mathcal{D}\ (\lambda y\ .\ (1+\varepsilon_a) \times y)\ 2))\\
&\quad= \mathcal{E}\ \varepsilon_a\ ((1+\varepsilon_a) \times (\mathcal{E}\ \varepsilon_b\ ((\lambda y\ .\ (1+\varepsilon_a) \times y)\ (2+\varepsilon_b))))\\
&\quad= \mathcal{E}\ \varepsilon_a\ ((1+\varepsilon_a) \times (\mathcal{E}\ \varepsilon_b\ ((1+\varepsilon_a) \times (2+\varepsilon_b))))\\
&\quad= \mathcal{E}\ \varepsilon_a\ ((1+\varepsilon_a) \times (\mathcal{E}\ \varepsilon_b\ ((2+2\varepsilon_a) + (1+\varepsilon_a)\varepsilon_b)))\\
&\quad= \mathcal{E}\ \varepsilon_a\ ((1+\varepsilon_a) \times (1+\varepsilon_a))\\
&\quad= \mathcal{E}\ \varepsilon_a\ (1+2\varepsilon_a)\\
&\quad= 2
\end{aligned}$$

where ε_a and ε_b are introduced by the two distinct invocations of $\mathcal{D}$. The accessor $\mathcal{E}$ is defined as

$$\mathcal{E}\ \varepsilon\ (x + x'\varepsilon) \stackrel{\triangle}{=} x'$$

and then $\mathcal{D}$ is defined as

$$\mathcal{D}\ f\ c \stackrel{\triangle}{=} \mathcal{E}\ \varepsilon\ (f\ (c+\varepsilon))$$

in which ε is unique to each live invocation of $\mathcal{D}$. As can be seen in the above example, failing to distinguish ε_a from ε_b would lead to an incorrect result: $(1+\varepsilon_a) \times (2+\varepsilon_b)$ would be interpreted as $(1+\varepsilon) \times (2+\varepsilon) = 2 + 3\varepsilon$ causing the above expression to evaluate to 3 instead of 2. Furthermore, even if we would distinguish ε_a from ε_b but erroneously take $\varepsilon_a \times \varepsilon_b = 0$ in a fashion analogous to $\varepsilon_a^2 = \varepsilon_b^2 = 0$ we would also obtain an incorrect result: $(1+\varepsilon_a) \times (2+\varepsilon_b)$ would reduce to $2 + 2\varepsilon_a + \varepsilon_b$ causing the above expression to evaluate to 1 instead of 2. Any implementation that did not posses a mechanism for properly distinguishing perturbations for different invocations of $\mathcal{D}$ or that failed to preserve nonzero cross perturbations could not support nested invocation of $\mathcal{D}$ or nested invocation of functions like MIN that utilize $\mathcal{D}$.

3 An API for Dual Numbers

As we have seen, nested invocations of $\mathcal{D}$ require distinct ε values. The components of a dual-number type created for a non-nested invocation of $\mathcal{D}$ are reals, while the components of a dual-number type created for a nested invocation of $\mathcal{D}$ are members of the dual-number type of the immediately surrounding invocation of $\mathcal{D}$. If "multiplied out," the resulting dual numbers correspond to first-order multinomials where the ε

values play the role of variables. This can be seen as a table of real numbers indexed by subsets of the live ε values. If the original nested structure is retained, we have a tree representation of depth n when there are n nested invocations of $\mathcal{D}$, with each level splitting on the presence of a particular ε value in the key, and the fringe holding the real numbers. Such tree representations are tempting because perturbations are often zero, and trees admit to a sparser representation where levels corresponding to perturbations of zero are skipped.

We impose an ordering on the ε values such that if ε is generated by an invocation of $\mathcal{D}$ nested inside the invocation of $\mathcal{D}$ that generated ε', then $\varepsilon' \prec \varepsilon$. Trees representing dual numbers can then obey the invariant that in a dual number $x + x'\varepsilon$ the x and x' slots are either reals or dual numbers over some ε' where $\varepsilon' \prec \varepsilon$, which improves efficiency. This is maintained in exhibited code, but made use of only in tree-based implementations of the following API for manipulating dual numbers:

$\textsc{DualNumber}?\ p$ returns true iff p is a dual number.
$\textsc{DualNumber}\ \varepsilon\ x\ 0 \stackrel{\triangle}{=} x$
$\textsc{DualNumber}\ \varepsilon\ x\ x' \stackrel{\triangle}{=} x + x'\varepsilon$
$\textsc{Epsilon}\ x + x'\varepsilon \stackrel{\triangle}{=} \varepsilon$
$\textsc{Primal}\ \varepsilon\ x \stackrel{\triangle}{=} x$ when x is a real.
$\textsc{Primal}\ \varepsilon\ (x + x'\varepsilon) \stackrel{\triangle}{=} x$
$\textsc{Primal}\ \varepsilon\ (x + x'\varepsilon') \stackrel{\triangle}{=} x + x'\varepsilon'$ when $\varepsilon' \prec \varepsilon$.
$\textsc{Perturbation}\ \varepsilon\ x \stackrel{\triangle}{=} 0$ when x is a real.
$\textsc{Perturbation}\ \varepsilon\ (x + x'\varepsilon) \stackrel{\triangle}{=} x'$
$\textsc{Perturbation}\ \varepsilon\ (x + x'\varepsilon') \stackrel{\triangle}{=} 0$ when $\varepsilon' \prec \varepsilon$.
$\textsc{Generate}_\varepsilon$ returns a fresh ε such that all other live $\varepsilon' \prec \varepsilon$.

Figure 1 contains an implementation of this API in SCHEME.[4] Note that the pattern of usage,[5] together with the above invariant, imply that $\textsc{Primal}\ \varepsilon\ (x + x'\varepsilon')$ and $\textsc{Perturbation}\ \varepsilon\ (x + x'\varepsilon')$ will never be called when $\varepsilon \prec \varepsilon'$.

4 An Implementation of $\mathcal{D}$ that Supports Nesting

Computing derivatives with dual numbers requires extensions of the arithmetic primitives. For instance

$$(x + x'\varepsilon) + (y + y'\varepsilon) = (x + y) + (x' + y')\varepsilon$$

Similarly, since $\varepsilon^2 = 0$

$$(x + x'\varepsilon) \times (y + y'\varepsilon) = (x \times y) + (x \times y' + x' \times y)\varepsilon$$

[4] All code examples from this paper are available from `http://www.bcl.hamilton.ie/~qobi/nesting/`.

[5] PRIMAL and PERTURBATION are only called in the definitions of `lift-real->real`, `lift-real*real->real`, and `primal*` in figure 2 and in the variant definitions of `derivative` on pages 6–9. In `lift-real->real` and `primal*`, all calls pass the ε of the second argument as the first argument. In `lift-real*real->real`, all calls pass the maximum ε of `p1` and `p2` as the second argument. In `derivative`, the call passes the generated ε for that invocation as the second argument.

```
(define <_e <)

(define dual-number?
 (let ((pair? pair?))
  (lambda (p) (and (pair? p) (eq? (car p) 'dual-number)))))

(define (dual-number e x x-prime)
 (if (zero? x-prime) x (list 'dual-number e x x-prime)))

(define epsilon cadr)

(define (primal e p)
 (if (or (not (dual-number? p)) (<_e (epsilon p) e)) p (caddr p)))

(define (perturbation e p)
 (if (or (not (dual-number? p)) (<_e (epsilon p) e)) 0 (cadddr p)))

(define generate-epsilon (let ((e 0)) (lambda () (set! e (+ e 1)) e)))
```

Fig. 1 A SCHEME implementation of the proposed API for dual numbers.

Note that the x, x', y, and y' values in the above might themselves be dual numbers with a different ε' generated from an earlier invocation of $\mathcal{D}$ than that which generated ε.

In the general case, a unary function $f : \alpha \to \alpha$ with derivative $f' : \alpha \to \alpha$ is extended to operate on dual numbers whose components are of type α as follows:

$$f\ (x + x'\varepsilon) = (f\ x) + ((f'\ x) \times x')\varepsilon$$

where $\times : \alpha \times \alpha \to \alpha$. Similarly, a binary function $f : \alpha \times \alpha \to \alpha$ whose derivatives with respect to the first and second arguments are $f_1 : \alpha \times \alpha \to \alpha$ and $f_2 : \alpha \times \alpha \to \alpha$ respectively is extended to operate on dual numbers whose components are of type α as follows:

$$f\ (x + x'\varepsilon)\ (y + y'\varepsilon) = (f\ x\ y) + ((f_1\ x\ y) \times x' + (f_2\ x\ y) \times y')\varepsilon$$

where $\times : \alpha \times \alpha \to \alpha$ and $+ : \alpha \times \alpha \to \alpha$. The SCHEME code in figure 2 implements the above mechanism in a fashion that will generate variants of functions that accept arguments of any dual-number type in the hierarchy and will automatically coerce elements of a lower type in the hierarchy to a higher type, as necessary, and treat native SCHEME numbers as elements of the base type in the hierarchy. Figure 3 contains code that uses the code in figure 2 to overload some numeric SCHEME primitives.

Given the code in figures 1, 2, and 3, a version of $\mathcal{D}$ that supports nesting can be implemented as:

```
(define (derivative f)
 (lambda (x)
  (let ((e (generate-epsilon)))
   (perturbation e (f (dual-number e x 1))))))
```

The above exposition demonstrates how to implement $\mathcal{D}$ as a referentially transparent defined function that allows nested invocation, in a purely functional style, through the use of a single non-referentially-transparent mechanism: the side effect in GENERATE$_\varepsilon$.

```
(define (lift-real->real f df/dx)
 (letrec ((self (lambda (p)
                 (if (dual-number? p)
                     (let ((e (epsilon p)))
                      (dual-number
                       e
                       (self (primal e p))
                       (* (df/dx (primal e p)) (perturbation e p))))
                     (f p)))))
  self))

(define (lift-real*real->real f df/dx1 df/dx2)
 (letrec ((self
           (lambda (p1 p2)
            (if (or (dual-number? p1)
                    (dual-number? p2))
                (let ((e (if (or (not (dual-number? p1))
                                 (and (dual-number? p2)
                                      (<_e (epsilon p1) (epsilon p2))))
                             (epsilon p2)
                             (epsilon p1))))
                 (dual-number
                  e
                  (self (primal e p1) (primal e p2))
                  (+ (* (df/dx1 (primal e p1) (primal e p2))
                        (perturbation e p1))
                     (* (df/dx2 (primal e p1) (primal e p2))
                        (perturbation e p2)))))
                (f p1 p2)))))
  self))

(define (primal* p)
 (if (dual-number? p) (primal* (primal (epsilon p) p)) p))

(define (lift-real^n->boolean f) (lambda ps (apply f (map primal* ps))))
```

Fig. 2 A mechanism for extending SCHEME procedures of type $\mathbb{R} \rightarrow \mathbb{R}$, $\mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$, and $\mathbb{R}^n \rightarrow$ **boolean** to support dual numbers.

5 Alternate Mechanisms for Generating Epsilons

One can implement a $\mathcal{D}$ that allows nested invocation by using a non-referentially-transparent mechanism to generate a new ε for each invocation of $\mathcal{D}$. The implementation in figure 1 represents ε values as integers and generates new ones using a non-referentially-transparent side-effect mechanism to increment a global counter.

Whenever a dual number with a non-zero perturbation of ε cannot escape an invocation of $\mathcal{D}$ that generates ε, the number of live ε values is bounded by the number of live invocations of $\mathcal{D}$. This is guaranteed to be the case when one refrains from using non-referentially-transparent language features, like side effects, dynamic scoping, locatives, generative types, `eq?`, `fluid-let`, `call/cc`, `dynamic-wind`, `throw`, `catch`, `block`, `return-from`, `unwind-protect`, etc., except to implement $\mathcal{D}$. In such cases, one can fold the generation of ε values into $\mathcal{D}$ as follows:

```
(define pair?
 (let ((pair? pair?))
  (lambda (x) (and (pair? x) (not (dual-number? x))))))

(define + (lift-real*real->real + (lambda (x1 x2) 1) (lambda (x1 x2) 1)))

(define - (lift-real*real->real - (lambda (x1 x2) 1) (lambda (x1 x2) -1)))

(define *
 (lift-real*real->real * (lambda (x1 x2) x2) (lambda (x1 x2) x1)))

(define /
 (lift-real*real->real
  / (lambda (x1 x2) (/ 1 x2)) (lambda (x1 x2) (- 0 (/ x1 (* x2 x2))))))

(define sqrt (lift-real->real sqrt (lambda (x) (/ 1 (* 2 (sqrt x))))))

(define exp (lift-real->real exp (lambda (x) (exp x))))

(define log (lift-real->real log (lambda (x) (/ 1 x))))

(define sin (lift-real->real sin (lambda (x) (cos x))))

(define cos (lift-real->real cos (lambda (x) (- 0 (sin x)))))

(define atan (lift-real*real->real
              atan
              (lambda (x1 x2) (/ (- 0 x2) (+ (* x1 x1) (* x2 x2))))
              (lambda (x1 x2) (/ x1 (+ (* x1 x1) (* x2 x2))))))

(define = (lift-real^n->boolean =))

(define < (lift-real^n->boolean <))

(define > (lift-real^n->boolean >))

(define <= (lift-real^n->boolean <=))

(define >= (lift-real^n->boolean >=))

(define zero? (lift-real^n->boolean zero?))

(define positive? (lift-real^n->boolean positive?))

(define negative? (lift-real^n->boolean negative?))

(define real? (lift-real^n->boolean real?))
```

Fig. 3 Overloading some SCHEME procedures that operate on reals with extensions that support dual numbers. Note that the overloaded `+`, `-`, `*`, `/`, and `atan` procedures are restricted to accept precisely two arguments.

```
(define derivative
 (let ((e 0))
  (lambda (f)
   (lambda (x)
    (set! e (+ e 1))
    (let ((result
           (perturbation e (f (dual-number e x 1)))))
     (set! e (- e 1))
     result)))))
```

Alternatively, one can replace one non-referentially-transparent mechanism, side effects, with another non-referentially-transparent mechanism, dynamic scoping via `fluid-let`, which mutates a variable for a constrained dynamic extent. This can generate distinct ε values for distinct dynamically nested invocations of $\mathcal{D}$.

```
(define derivative
 (let ((e 0))
  (lambda (f)
   (lambda (x)
    (fluid-let ((e (+ e 1)))
     (perturbation e (f (dual-number e x 1))))))))
```

When, additionally, the implementation uses a stack for activation records and it can be guaranteed that activation records corresponding to nested function invocations will be allocated at increasing addresses, one can alternatively use another non-referentially-transparent mechanism, locatives:

```
(define (derivative f)
 (lambda (x)
  (let ((e (variable-address->integer x)))
   (perturbation e (f (dual-number e x 1))))))
```

In this variation, the alpha renaming that is performed by a typical programming-language implementation as part of beta reduction distinguishes ε values generated by distinct invocations.

An alternative to representing dual numbers as explicit trees would be to represent their fringe as a (potentially sparse) association list indexed by path. For example, the nested dual-number tree

$$((2 + 2\varepsilon_a) + (1 + \varepsilon_a)\varepsilon_b)$$

can be multiplied out as

$$2 + 2\varepsilon_a + \varepsilon_b + \varepsilon_a\varepsilon_b$$

which would be represented as the association list

$$\{\{\} \mapsto 2, \{\varepsilon_a\} \mapsto 2, \{\varepsilon_b\} \mapsto 1, \{\varepsilon_a, \varepsilon_b\} \mapsto 1\}$$

This strategy eliminates the need for ε values to be ordered by invocation depth, thus admitting an implementation where ε values are unique but not ordered. An implementation of our API for dual numbers that uses such a representation is shown in figure 4. This implements $\mathcal{D}$ where ε values are represented as fresh pairs allocated by `cons`, a referentially-transparent mechanism, in concert with `eq?`, a non-referentially-transparent mechanism, and is reminiscent of a (non-referentially-transparent) technique used in HASKELL called observable sharing (Claessen and Sands, 1999).

Yet another alternative strategy for representing dual numbers is to represent the ε values implicitly as types instead of explicitly as integers, using another non-referentially-transparent mechanism, generative structure types, such as those available in PLT SCHEME (Flatt, 2005). An implementation of this strategy is given in figure 5.

As noted by Alex Shafarenko (personal communication), the need to distinguish the different ε values introduced by different invocations of $\mathcal{D}$ is similar, in some ways, to the need to distinguish different lambda-bound variables with the same name during beta reduction to avoid capturing free variables. The latter is accomplished via the alpha renaming that is performed by a typical programming-language implementation. However, as noted above, the ε values are not represented as programming-language variables, since dual numbers are represented as data structures, not terms. Thus the

```
(define (<_e e1 e2) #t)

(define (some p l)
 (and (not (null? l)) (or (p (car l)) (some p (cdr l)))))

(define (find-if p l)
 (let loop ((l l))
  (cond ((null? l) #f)
        ((p (car l)) (car l))
        (else (loop (cdr l))))))

(define (remove-if p l)
 (let loop ((l l) (c '()))
  (cond ((null? l) (reverse c))
        ((p (car l)) (loop (cdr l) c))
        (else (loop (cdr l) (cons (car l) c))))))

(define (removeq x l)
 (let loop ((l l) (c '()))
  (cond ((null? l) (reverse c))
        ((eq? x (car l)) (loop (cdr l) c))
        (else (loop (cdr l) (cons (car l) c))))))

(define terms
 (let ((pair? pair?))
  (lambda (p)
   (if (and (pair? p) (eq? (car p) 'dual-number))
       (cadr p)
       (list (cons '() p))))))

(define (terms->dual-number terms)
 (cond ((null? terms) 0)
       ((and (null? (cdr terms)) (null? (car (car terms))))
        (cdr (car terms)))
       (else (list 'dual-number terms))))

(define (dual-number? p)
 (some (lambda (term) (not (null? (car term)))) (terms p)))

(define (dual-number e x x-prime)
 (terms->dual-number
  (append (terms x)
          (map (lambda (term) (cons (cons e (car term)) (cdr term)))
               (terms x-prime)))))

(define (epsilon p)
 (car (car (find-if (lambda (term) (not (null? (car term)))) (terms p)))))

(define (primal e p)
 (terms->dual-number
  (remove-if (lambda (term) (memq e (car term))) (terms p))))

(define (perturbation e p)
 (terms->dual-number
  (map (lambda (term) (cons (removeq e (car term)) (cdr term)))
       (remove-if (lambda (term) (not (memq e (car term)))) (terms p)))))

(define (generate-epsilon) (cons #f #f))
```

Fig. 4 Implementation of an alternate representation for dual numbers as sparse association lists of their fringe elements indexed by path.

typical mechanism of alpha-renaming does not suffice to implement a $\mathcal{D}$ that allows nested invocation.

Rewrite systems are often formulated in terms of rules that map source-term patterns to target-term patterns. Such term patterns may contain pattern variables that

```
(define (derivative f)
 (lambda (x)
  (let-struct bundle (primal tangent)
   (define (dual-number x x-prime)
    (if (zero? x-prime) x (make-bundle x x-prime)))

   (define (primal p) (if (bundle? p) (bundle-primal p) p))

   (define (perturbation p) (if (bundle? p) (bundle-tangent p) 0))

   (define (raise-alpha->alpha f df/dx)
    (let ((* *))
     (lambda (p)
      (dual-number
       (f (primal p)) (* (df/dx (primal p)) (perturbation p))))))

   (define (raise-alpha*alpha->alpha f df/dx1 df/dx2)
    (let ((+ +) (* *))
     (lambda (p1 p2)
      (dual-number
       (f (primal p1) (primal p2))
       (+ (* (df/dx1 (primal p1) (primal p2)) (perturbation p1))
          (* (df/dx2 (primal p1) (primal p2)) (perturbation p2)))))))

   (define (raise-alpha^n->boolean f)
    (lambda ps (apply f (map primal ps))))

   (fluid-let ((+ (raise-alpha*alpha->alpha
                   + (lambda (x1 x2) 1) (lambda (x1 x2) 1)))
               (- (raise-alpha*alpha->alpha
                   - (lambda (x1 x2) 1) (lambda (x1 x2) -1)))
               (* (raise-alpha*alpha->alpha
                   * (lambda (x1 x2) x2) (lambda (x1 x2) x1)))
               (/ (let ((- -) (* *) (/ /))
                   (raise-alpha*alpha->alpha
                    /
                    (lambda (x1 x2) (/ 1 x2))
                    (lambda (x1 x2) (- 0 (/ x1 (* x2 x2)))))))
               (sqrt (let ((* *) (/ /) (sqrt sqrt))
                      (raise-alpha->alpha
                       sqrt (lambda (x) (/ 1 (* 2 (sqrt x)))))))
               (exp (raise-alpha->alpha exp exp))
               (log (let ((/ /))
                     (raise-alpha->alpha log (lambda (x) (/ 1 x)))))
               (sin (raise-alpha->alpha sin cos))
               (cos (let ((- -) (sin sin))
                     (raise-alpha->alpha cos (lambda (x) (- 0 (sin x))))))
               (atan (let ((+ +) (- -) (* *) (/ /))
                      (raise-alpha*alpha->alpha
                       atan
                       (lambda (x1 x2)
                        (/ (- 0 x2) (+ (* x1 x1) (* x2 x2))))
                       (lambda (x1 x2) (/ x1 (+ (* x1 x1) (* x2 x2)))))))
               (= (raise-alpha^n->boolean =))
               (< (raise-alpha^n->boolean <))
               (> (raise-alpha^n->boolean >))
               (<= (raise-alpha^n->boolean <=))
               (>= (raise-alpha^n->boolean >=))
               (zero? (raise-alpha^n->boolean zero?))
               (positive? (raise-alpha^n->boolean positive?))
               (negative? (raise-alpha^n->boolean negative?))
               (real? (raise-alpha^n->boolean real?)))
    (perturbation (f (dual-number x 1)))))))
```

Fig. 5 Implementation of $\mathcal{D}$ in PLT Scheme using generative structure types.

range over terms. If a pattern variable in the target pattern appears in the source pattern, it is bound, during rewrite, to the subterm matching the pattern variable in the source term. If a pattern variable in the target pattern does not appear in the source pattern, it is free. Some rewrite systems take free pattern variables to denote the generation of a fresh variable in the term language. This constitutes a form of alpha renaming. Unrestricted use of such a facility would not be referentially transparent. However, one can formulate a $\mathcal{D}$ that is referentially transparent and that allows nested invocation as a rewrite rule in such a rewrite system

$$\mathcal{D}\ f\ c \rightsquigarrow \mathcal{E}\ \varepsilon\ (f\ (c+\varepsilon))$$

where f, c, and ε are pattern variables.

A similar capability exists in Prolog. Variables in the right-hand side of a clause that do not appear in the left-hand side generate logic variables. These are implemented in a distinct fashion from those that do appear in the left-hand side. Proper implementation requires both kinds of variables to be alpha renamed during resolution. Pure Prolog, including logic variables and their requisite alpha renaming, is referentially transparent. However, implementing a $\mathcal{D}$ that uses logic variables to distinguish ε values requires the use of a non-referentially-transparent extra-logical primitive to prevent unification of such logic variables.

6 Eliminating Run-Time Generation of ε Values

Implementing a $\mathcal{D}$ that allows nested invocation requires that each nested invocation of $\mathcal{D}$ have a new ε value. This can be done dynamically using a single non-referentially-transparent mechanism. However static mechanisms can be used instead under special circumstances, namely when static analysis can determine sufficient information about the dynamic call graph involving $\mathcal{D}$ to allow static allocation of ε values.

The static analyses and transformations can be manually simulated by a programmer. To do this, one must expose ε as a parameter to $\mathcal{D}$

$$\mathcal{D}\ \varepsilon\ f\ c \stackrel{\triangle}{=} \mathcal{E}\ \varepsilon\ (f\ (c+\varepsilon))$$

and require the programmer to guarantee that each nested invocation of $\mathcal{D}$ is supplied with a distinct ε and that these obey the $\prec$ invariant. In the general case, this requires that each function, such as Min, that calls $\mathcal{D}$, directly or indirectly, also expose ε as a parameter. This would be a serious violation of modularity and separation of concerns: in general, the *caller* of a higher-order function like Min should be oblivious to whether or not that higher-order function uses $\mathcal{D}$ internally. Such a discipline would also make expressions involving the $\mathcal{D}$ operator extremely fragile.

7 Example

The ability to nest invocation of $\mathcal{D}$ is useful in numerical simulation of physical systems, as is illustrated by the following example. Consider a charged particle traveling non-relativistically in a plane with position $\mathbf{x}(t)$, velocity $\dot{\mathbf{x}}(t)$, initial position $\mathbf{x}(0)=(0,8)$, and initial velocity $\dot{\mathbf{x}}(0) = (0.75, 0)$. It is accelerated by an electric field formed by a pair of repulsive bodies,

$$p(\mathbf{x}; w) = \|\mathbf{x} - (10, 10 - w)\|^{-1} + \|\mathbf{x} - (10, 0)\|^{-1}$$

where w is a modifiable control parameter of the system. The particle hits the x-axis at position $\mathbf{x}(t_f)$. We use a textbook implementation of Newton's method to optimize w so as to minimize $E(w) = x_0(t_f)^2$, with the goal of finding a value for w that causes the particle's path to intersect the origin.

We use Naive Euler ODE integration

$$\begin{aligned}\ddot{\mathbf{x}}(t) &= -\left.\nabla_{\mathbf{x}}\, p(\mathbf{x})\right|_{\mathbf{x}=\mathbf{x}(t)}\\ \dot{\mathbf{x}}(t + \Delta t) &= \dot{\mathbf{x}}(t) + \Delta t\, \ddot{\mathbf{x}}(t)\\ \mathbf{x}(t + \Delta t) &= \mathbf{x}(t) + \Delta t\, \dot{\mathbf{x}}(t)\end{aligned}$$

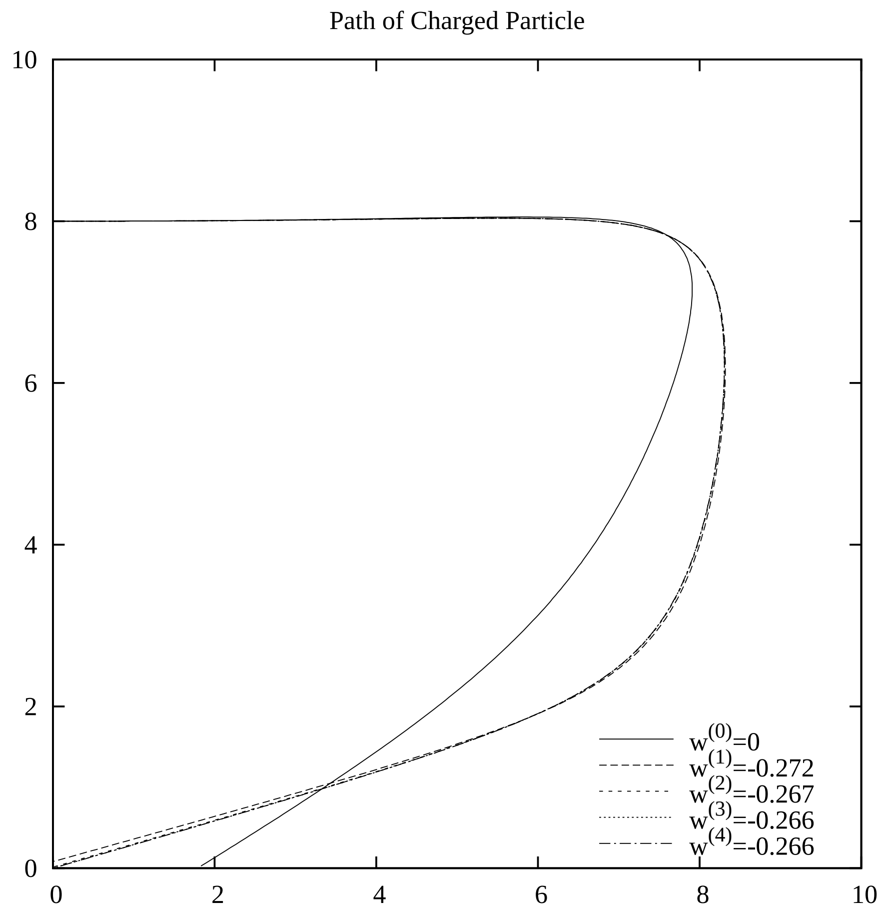

Fig. 6 Plot of the path of a charged particle at various points during Newton optimization of the parameter w controlling an electric field to minimize the distance between the particle's x-intercept and the origin.

```
(define first car)

(define rest cdr)

(define (map-n f n)
 (let loop ((i 0)) (if (= i n) '() (cons (f i) (loop (+ i 1))))))

(define (reduce f l i)
 (if (null? l) i (f (first l) (reduce f (rest l) i))))

(define (sqr x) (* x x))

(define (v+ u v) (map + u v))

(define (v- u v) (map - u v))

(define (k*v k v) (map (lambda (x) (* k x)) v))

(define (dot u v) (reduce + (map * u v) 0))

(define (distance u v) (let ((d (v- v u))) (sqrt (dot d d))))

(define (replace-ith x i xi)
 (if (zero? i)
     (cons xi (rest x))
     (cons (first x) (replace-ith (rest x) (- i 1) xi))))

(define (gradient f)
 (lambda (x)
  (map-n
   (lambda (i)
    ((derivative (lambda (xi) (f (replace-ith x i xi)))) (list-ref x i)))
   (length x))))

(define x-initial '(0 8))
(define xdot-initial '(0.75 0))
(define w0 0)
(define error-tolerance 1e-1)
(define delta-t 1e-1)

(define (naive-euler w)
 (let ((charges (list (list 10 (- 10 w)) (list 10 0))))
  (define (p x)
   (reduce + (map (lambda (c) (/ 1 (distance x c))) charges) 0))
  (let loop ((x x-initial) (xdot xdot-initial))
   (let* ((xddot (k*v -1 ((gradient p) x)))
          (x-new (v+ x (k*v delta-t xdot))))
    (if (positive? (list-ref x-new 1))
        (loop x-new (v+ xdot (k*v delta-t xddot)))
        (let* ((delta-t-f (/ (- 0 (list-ref x 1)) (list-ref xdot 1)))
               (x-t-f (v+ x (k*v delta-t-f xdot))))
         (sqr (list-ref x-t-f 0))))))))

(define (argmin-using-textbook-newtons-method f x)
 (let loop ((x x) (i 0))
  (let ((df-dx ((derivative f) x)))
   (if (< (abs df-dx) error-tolerance)
       x
       (loop (- x (/ df-dx ((derivative (derivative f)) x))) (+ i 1))))))

(define (particle) (argmin-using-textbook-newtons-method naive-euler w0))
```

Fig. 7 An abbreviated version of the code that implements the charged particle path-optimization example from section 7. The unabbreviated code that produced figure 6 is available at `http://www.bcl.hamilton.ie/~qobi/nesting/`.

to compute the particle's path, taking $\Delta t = 10^{-1}$. We use linear interpolation to find the point where the particle hits the x-axis.

$$\begin{aligned}
\text{When } x_1(t+\Delta t) &\leq 0 \\
\text{let: } \Delta t_f &= -\frac{x_1(t)}{\dot{x}_1(t)} \\
t_f &= t + \Delta t_f \\
\mathbf{x}(t_f) &= \mathbf{x}(t) + \Delta t_f \dot{\mathbf{x}}(t) \\
\text{Error: } E(w) &= x_0(t_f)^2
\end{aligned}$$

We use $\mathcal{D}$ to calculate $\nabla_{\mathbf{x}}\, p(\mathbf{x})$ and also to calculate the first and second derivatives of E with respect to w when minimizing E using Newton's method.

$$w^{(i+1)} = w^{(i)} - \frac{E'(w^{(i)})}{E''(w^{(i)})}$$

Note that computing E invokes $\mathcal{D}$ to compute $\nabla_{\mathbf{x}}\, p(\mathbf{x})$ and thus computing E' and E'' involve nested invocation of $\mathcal{D}$. We start the minimization process at $w^{(0)} = 0$ and terminate the minimization when $|E'(w^{(i)})| < 10^{-1}$. The paths taken by the particle at each iteration of the minimization process are shown in figure 6. Code that implements this example is given in figure 7.

8 Discussion

It is quite natural to consider augmenting a functional-programming language with a derivative-taking operator like $\mathcal{D}$. Indeed, derivative-taking operators were used as a motivation for the lambda calculus.

> It is, of course, not excluded that the range of arguments or range of values of a function should consist wholly or partly of functions. The derivative, as this notion appears in the elementary differential calculus, is a familiar mathematical example of a function for which both ranges consist of functions. (Church, 1941, ¶4)

We have taken this example to heart and explored issues that arise when implementing $\mathcal{D}$, a derivative-taking operator that uses forward-mode AD. Interestingly, we found no way to implement $\mathcal{D}$ in a pure lambda calculus, and a simple example[6] seems to show that $\mathcal{D}$ cannot be formulated in Church's original untyped lambda calculus. We were, however, able to implement $\mathcal{D}$, which is itself pure, using any one of a variety of impure mechanisms.

Techniques roughly similar to those in figure 4 were used to implement a nestable version of $\mathcal{D}$ in the undocumented internals of SCMUTILS, a software package accompanying a textbook on classical mechanics (Sussman et al., 2001). On the other hand, previous implementations of forward-mode AD in pure HASKELL (Karczmarczuk, 1998a,b,

[6] Consider $\mathcal{D}\ (\lambda x\ .\ x \times (\mathcal{D}\ (\lambda y\ .\ \boxed{x})\ c))\ c$ versus $\mathcal{D}\ (\lambda x\ .\ x \times (\mathcal{D}\ (\lambda y\ .\ \boxed{y})\ c))\ c$. In the untyped lambda calculus, the boxed x and y must have the same value, since the only operation $\mathcal{D}$ can perform on the function it receives as its first argument is to call it with some argument. That being so, the inner calls to $\mathcal{D}$ in the two cases evaluate to the same value. But for the results to be correct (0 for the expression on the left, 1 for that on the right), the inner calls must evaluate to correct values, with the inner call on the left evaluating to 0 and that on the right evaluating to 1.

1999, 2001; Nilsson, 2003) do not include mechanisms that would support implementation of a nestable $\mathcal{D}$. Indeed, we hypothesize that a nestable $\mathcal{D}$ *cannot* be formulated as a function definition in current pure dialects of HASKELL.

While all known techniques for implementing a nestable $\mathcal{D}$ use non-referentially-transparent mechanisms, $\mathcal{D}$ itself *is* referentially transparent. This motivates inclusion of $\mathcal{D}$, or similar functionality, as a primitive feature of pure functional-programming languages whose intended uses include numeric computing.

Acknowledgements The authors thank Jerzy Karczmarczuk for many interesting discussions, and the anonymous reviewers for their valuable suggestions.

References

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

Claessen, K. and D. Sands: 1999, 'Observable Sharing for Functional Circuit Description'. In: *Proc. of Asian Computer Science Conference (ASIAN)*. Springer-Verlag.

Clifford, W. K.: 1873, 'Preliminary Sketch of Bi-quaternions'. *Proceedings of the London Mathematical Society* **4**, 381–95.

Flatt, M.: 2005, 'PLT MzScheme: Language Manual'. Technical Report PLT-TR05-1-v300, PLT Scheme Inc.

Karczmarczuk, J.: 1998a, 'Functional Differentiation of Computer Programs'. In: *Proceedings of the III ACM SIGPLAN International Conference on Functional Programming*. Baltimore, MD, pp. 195–203.

Karczmarczuk, J.: 1998b, 'Lazy Differential Algebra and its Applications'. In: *Workshop, III International Summer School on Advanced Functional Programming*. Braga, Portugal.

Karczmarczuk, J.: 1999, 'Functional Coding of Differential Forms'. In: *Scottish Workshop on FP*.

Karczmarczuk, J.: 2001, 'Functional Differentiation of Computer Programs'. *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 quantities, and a remarkable type of calculus for this'. *Acta Eruditorum*.

Newton, I.: 1704, 'De quadratura 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*. Uppsala, Sweden, pp. 153–64, ACM Press.

Pearlmutter, B. A. and J. M. Siskind: 2007, 'Lazy Multivariate Higher-Order Forward-Mode AD'. In: *Proceedings of the 2007 Symposium on Principles of Programming Languages*. Nice, France, pp. 155–60.

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

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