To appear in POPL 2007

Lazy Multivariate Higher-Order Forward-Mode AD

Barak A. Pearlmutter
Hamilton Institute
NUI Maynooth, Ireland
barak@cs.nuim.ie

Jeffrey Mark Siskind
School of Electrical and Computer Engineering
Purdue University, USA
qobi@purdue.edu

Abstract

A method is presented for computing all higher-order partial derivatives of a multivariate function $\mathbb{R}^n \rightarrow \mathbb{R}$. This method works by evaluating the function under a nonstandard interpretation, lifting reals to multivariate power series. Multivariate power series, with potentially an infinite number of terms with nonzero coefficients, are represented using a lazy data structure constructed out of linear terms. A complete implementation of this method in SCHEME is presented, along with a straightforward exposition, based on Taylor expansions, of the method's correctness.

Categories and Subject Descriptors G.1.4 [*Quadrature and Numerical Differentiation*]: Automatic differentiation; D.3.2 [*Language Classifications*]: Applicative (functional) languages

General Terms Algorithms, Languages

Keywords Power series, Nonstandard interpretation

1. Introduction

Forward-Mode Automatic Differentiation, or forward AD, [1] is a method for adapting a program that computes a function to yield one that computes its derivatives. Karczmarczuk [2–5] presented an implementation of forward AD in HASKELL. This implementation had a novel characteristic: it adapted a program that computed a univariate function $f : \mathbb{R} \rightarrow \mathbb{R}$ to yield one that produced an infinite stream of higher-order derivatives $(f(x), f'(x), f''(x), \ldots)$. However, Karczmarczuk provided the details for his method only for univariate functions. Karczmarczuk [4] hinted at a generalization to multivariate functions but did not provide the details. Here, we present the details of a novel generalization of Karczmarczuk's method to the multivariate case. In part, we use methods previously developed for implementing nestable first-order forward AD in a functional framework [6]. The crucial additional insight here, both for developing the extension and for demonstrating its correctness, involves reformulating Karczmarczuk's method using Taylor expansions instead of the chain rule. This requires dealing with the requisite factorial factors.

2. Univariate First-Order Forward AD

The Taylor expansion [7] of $f(c+\varepsilon)$ with respect to ε is

$$f(c+\varepsilon) = \sum_{i=0}^{\infty} \frac{1}{i!} \left.\frac{\mathrm{d}^i f(x)}{\mathrm{d}x^i}\right|_{x=c} \varepsilon^i$$

This implies that one can compute the i-th derivative of a univariate function f at a scalar point c by evaluating by evaluating $f(c+\varepsilon)$ under a nonstandard interpretation replacing real numbers with univariate power series in ε, extracting the coefficient of ε^i in the result, and multiplying this by $i!$. Traditional forward AD [1] truncates the Taylor expansions at $i > 1$, thus computing a representation that contains only the first derivative.

Such truncated Taylor expansions are *dual numbers* [8]. We denote a dual number p as $x + x'\varepsilon$, by analogy with the standard notation $a + b\mathrm{i}$ for complex numbers. 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 $\varepsilon \neq 0$. Furthermore, just as implementations of complex arithmetic typically represent complex numbers $a + b\mathrm{i}$ as Argand pairs $\langle a, b\rangle$, implementations of forward AD typically represent dual numbers $x + x'\varepsilon$ as tangent-bundle pairs $\langle x, x'\rangle$. Finally, just as implementations of complex arithmetic typically overload the arithmetic primitives to manipulate complex numbers, implementations of forward AD typically overload the arithmetic primitives to manipulate dual numbers.

We use $\mathcal{E}\ \varepsilon\ p$ to denote the coefficient of ε in the dual number p

$$\mathcal{E}\ \varepsilon\ (x + x'\varepsilon) \stackrel{\triangle}{=} x' \tag{1}$$

and $\mathcal{D}\ f\ c$ to denote the value of the first derivative of a univariate function f at a scalar point c. Forward AD computes $\mathcal{D}\ f\ c$ by evaluating $f\ (c+\varepsilon)$ under a nonstandard interpretation replacing real numbers with dual numbers and extracting the coefficient of ε in the result.

$$\mathcal{D}\ f\ c \stackrel{\triangle}{=} \mathcal{E}\ \varepsilon\ (f\ (c+\varepsilon)) \tag{2}$$

The εs introduced by nested invocations of $\mathcal{D}$ must be distinct [6].

To see how this works, let us manually apply the mechanism to a simple example: computing the first derivative of $f(x) = x^4 + 2x^3$ at $x = 3$. To do this, we first evaluate $f(3+\varepsilon)$.

$$\begin{aligned} f(3+\varepsilon) &= (3+\varepsilon)^4 + 2(3+\varepsilon)^3 \\ &= (81 + 108\varepsilon) + 2(27 + 27\varepsilon) \\ &= 135 + 162\varepsilon \end{aligned}$$

From this we can extract the derivative: $\mathcal{E}\ \varepsilon\ (135 + 162\varepsilon) = 162$.

$$\left.\frac{\mathrm{d}f(x)}{\mathrm{d}x}\right|_{x=3} = \left. 4x^3 + 6x^2\right|_{x=3} = 162 = \mathcal{E}\ \varepsilon\ (f\ (3+\varepsilon))$$

Note that the above makes use of the restriction that $\varepsilon^2 = 0$ when evaluating $(3+\varepsilon)^3 = 27+27\varepsilon$ and $(3+\varepsilon)^4 = 81+108\varepsilon$, dropping

POPL'07 January 17–19, 2007, Nice, France.

the ε^2, ε^3, and ε^4 terms. This is the essence of traditional forward AD when limited to the case of univariate derivatives.

3. Univariate Higher-Order Forward AD

While the above nominally computes only first derivatives, straightforward repeated application of $\mathcal{D}$ allows computation of higher-order derivatives. We use $\mathcal{D}\ f$ to denote partial application of $\mathcal{D}$, i.e., $\lambda c\ .\ (\mathcal{D}\ f\ c)$, and $\mathcal{D}^i\ f\ c$ to denote the value of the i-th derivative of a univariate function f at a scalar point c.

$$\mathcal{D}^0\ f \quad \triangleq \quad f \tag{3}$$

$$\mathcal{D}^i\ f \quad \triangleq \quad \mathcal{D}^{i-1}\ (\mathcal{D}\ f) \ \ \text{when } i > 0 \tag{4}$$

We refer to the above method for computing higher-order derivatives as the *repetition* method.

Karczmarczuk [2–5] presented an alternate method for computing higher-order univariate derivatives. His method can be viewed as computing non-truncated Taylor expansions,[1] removing the restriction that $\varepsilon^2 = 0$, and generalizing dual numbers to univariate power series in ε. To accomplish this, we first extend the definition of $\mathcal{E}$ from (1) so that $\mathcal{E}\ \varepsilon^i\ p$ yields $i!$ times the coefficient of ε^i in the power series p.

$$\mathcal{E}\ \varepsilon^0\ p \quad \triangleq \quad \mathcal{R}\ \varepsilon\ p \tag{5}$$

$$\mathcal{E}\ \varepsilon^i\ p \quad \triangleq \quad i \times \mathcal{E}\ \varepsilon^{i-1}\ (\mathcal{Q}\ \varepsilon\ p) \ \ \text{when } i > 0 \tag{6}$$

In the above and throughout this paper, $\mathcal{Q}\ \varepsilon\ p$ and $\mathcal{R}\ \varepsilon\ p$ destructure a power series: $p = (\mathcal{R}\ \varepsilon\ p) + (\mathcal{Q}\ \varepsilon\ p)\varepsilon$. Given the above, Karczmarczuk's method can be viewed as computing $\mathcal{D}^i\ f\ c$ by evaluating $f\ (c + \varepsilon)$ under a nonstandard interpretation replacing real numbers with univariate power series in ε, extracting the coefficient of ε^i in the result, and multiplying this by $i!$.

$$\mathcal{D}^i\ f\ c \triangleq \mathcal{E}\ \varepsilon^i\ (f\ (c+\varepsilon)) \tag{7}$$

To see how this works, let us manually apply the mechanism to a simple example: computing all of the higher-order derivatives of $f(x) = x^4 + 2x^3$ at $x = 3$. To do this, we first evaluate $f(3 + \varepsilon)$.

$$\begin{aligned} f(3+\varepsilon) &= (3+\varepsilon)^4 + 2(3+\varepsilon)^3 \\ &= (81 + 108\varepsilon + 54\varepsilon^2 + 12\varepsilon^3 + \varepsilon^4) \\ &\quad + 2(27 + 27\varepsilon + 9\varepsilon^2 + \varepsilon^3) \\ &= 135 + 162\varepsilon + 72\varepsilon^2 + 14\varepsilon^3 + \varepsilon^4 \end{aligned}$$

From this we can extract all of the higher-order derivatives.

$$\begin{aligned} f(x)\Big|_{x=3} &= x^4 + 2x^3\Big|_{x=3} = 135 = 0! \times 135 = \mathcal{E}\ \varepsilon^0\ (f\ (3+\varepsilon)) \\ \frac{\mathrm{d}f(x)}{\mathrm{d}x}\Big|_{x=3} &= 4x^3 + 6x^2\Big|_{x=3} = 162 = 1! \times 162 = \mathcal{E}\ \varepsilon^1\ (f\ (3+\varepsilon)) \\ &\vdots \\ \frac{\mathrm{d}^4 f(x)}{\mathrm{d}x^4}\Big|_{x=3} &= \qquad 24\Big|_{x=3} = \ 24 = 4! \times 1 \quad = \mathcal{E}\ \varepsilon^4\ (f\ (3+\varepsilon)) \end{aligned}$$

4. Lazy Univariate Higher-Order Forward AD

The input to the nonstandard interpretation will always be a polynomial $c + \varepsilon$, an entity with a finite number of terms with nonzero coefficients. In the above example, the output of the nonstandard interpretation was also a polynomial. However some functions, such as $\sin$, yield a result entity with an infinite number of terms with nonzero coefficients, i.e., a power series, even when applied to an input polynomial. Karczmarczuk addressed this by representing univariate power series as lazy unidimensional streams. We refer to this as a *tower* method. Karczmarczuk presented the details of his tower method only for univariate functions. The remainder of this paper presents the details of a novel tower method that generalizes to the multivariate case.

5. Multivariate Higher-Order Forward AD

We use $\mathcal{D}^{(i_1,\ldots,i_n)}\ f\ (c_1,\ldots,c_n)$ to denote the value of

$$\left.\frac{\partial^{i_1+\cdots+i_n} f(x_1,\ldots,x_n)}{\partial {x_1}^{i_1} \cdots \partial {x_n}^{i_n}}\right|_{x_1=c_1,\ldots,x_n=c_n}$$

i.e., the value of a higher-order partial derivative of a multivariate function f at a multidimensional point $(c_1,\ldots,c_n)$. One can generalize the repetition method to the multivariate case.

$$\mathcal{D}^{(0,\ldots,0)}\ f \quad \triangleq \quad f \tag{8}$$

$$\begin{aligned} &\mathcal{D}^{(i_1,\ldots,i_n)}\ f \quad \triangleq \\ &\quad \mathcal{D}^{(i_1,\ldots,i_{\ell-1},i_\ell-1,i_{\ell+1},\ldots,i_n)} \\ &\quad \lambda(c_1,\ldots,c_n)\ .\ \mathcal{D}\ (\lambda u\ .\ f\ (c_1,\ldots,c_{\ell-1},u,c_{\ell+1},\ldots,c_n))\ c_\ell \\ &\quad \text{when } i_\ell > 0 \end{aligned} \tag{9}$$

Again, each nested invocation of $\mathcal{D}$ must use a distinct ε [6].

One can formulate a multivariate tower method by generalizing univariate power series to multivariate power series. To accomplish this, we first note that the multivariate Taylor expansion of $f((c_1+\varepsilon_1),\ldots,(c_n+\varepsilon_n))$ with respect to $(\varepsilon_1,\ldots,\varepsilon_n)$ is

$$\sum_{i_1=0}^{\infty} \cdots \sum_{i_n=0}^{\infty} \frac{1}{i_1! \cdots i_n!} \left.\frac{\partial^{i_1+\cdots+i_n} f(x_1,\ldots,x_n)}{\partial {x_1}^{i_1} \cdots \partial {x_n}^{i_n}}\right|_{x_1=c_1,\ldots,x_n=c_n} \varepsilon_1^{i_1} \cdots \varepsilon_n^{i_n}$$

We therefore extend the definition of $\mathcal{E}$ from (5–6) to the multivariate case so that $\mathcal{E}\ {\varepsilon_1}^{i_1} \cdots {\varepsilon_n}^{i_n}\ p$ yields $i_1! \cdots i_n!$ times the coefficient of $\varepsilon_1^{i_1} \cdots \varepsilon_n^{i_n}$ in the power series p.

$$\mathcal{E}\ 1\ p \quad \triangleq \quad p \tag{10}$$

$$\mathcal{E}\ {\varepsilon_1}^{i_1} \cdots {\varepsilon_n}^{i_n}\ p \quad \triangleq \quad \mathcal{E}\ {\varepsilon_2}^{i_2} \cdots {\varepsilon_n}^{i_n}\ (\mathcal{E}\ {\varepsilon_1}^{i_1}\ p) \tag{11}$$

Given the above, one can compute $\mathcal{D}^{(i_1,\ldots,i_n)}\ f\ (c_1,\ldots,c_n)$ by evaluating $f\ ((c_1+\varepsilon_1),\ldots,(c_n+\varepsilon_n))$ under a nonstandard interpretation replacing real numbers with multivariate power series in distinct $\varepsilon_1,\ldots,\varepsilon_n$, extracting the coefficient of ${\varepsilon_1}^{i_1} \cdots {\varepsilon_n}^{i_n}$ in the result, and multiplying this by $i_1! \cdots i_n!$.

$$\begin{aligned} &\mathcal{D}^{(i_1,\ldots,i_n)}\ f\ (c_1,\ldots,c_n) \quad \triangleq \\ &\qquad \mathcal{E}\ {\varepsilon_1}^{i_1} \cdots {\varepsilon_n}^{i_n}\ (f\ ((c_1+\varepsilon_1),\ldots,(c_n+\varepsilon_n))) \end{aligned} \tag{12}$$

In the above, the ε_ℓ must be distinct from each other and from any other ε used by nested invocations of any form of $\mathcal{D}$ [6].

To see how this works, let us manually apply the mechanism to a simple example: computing all of the higher-order partial derivatives of $g(x,y) = x^3y + x^2y^2$ at $x = 2$ and $y = 3$. To

[1] As discussed in section 7, this is a reformulation of Karczmarczuk's method which was originally formulated using the chain rule.

do this, we first evaluate $g((2+\varepsilon_x),(3+\varepsilon_y))$.

$$\begin{aligned}
&g((2+\varepsilon_x),(3+\varepsilon_y))\\
&\quad= (2+\varepsilon_x)^3(3+\varepsilon_y)+(2+\varepsilon_x)^2(3+\varepsilon_y)^2\\
&\quad= (8+12\varepsilon_x+6\varepsilon_x^2+\varepsilon_x^3)(3+\varepsilon_y)\\
&\qquad\quad+(4+4\varepsilon_x+\varepsilon_x^2)(9+6\varepsilon_y+\varepsilon_y^2)\\
&\quad= (24+36\varepsilon_x+8\varepsilon_y+18\varepsilon_x^2+12\varepsilon_x\varepsilon_y+3\varepsilon_x^3+6\varepsilon_x^2\varepsilon_y\\
&\qquad+\varepsilon_x^3\varepsilon_y)\\
&\qquad+(36+36\varepsilon_x+24\varepsilon_y+9\varepsilon_x^2+24\varepsilon_x\varepsilon_y+4\varepsilon_y^2+6\varepsilon_x^2\varepsilon_y\\
&\qquad\quad+4\varepsilon_x\varepsilon_y^2+\varepsilon_x^2\varepsilon_y^2)\\
&\quad= (60+72\varepsilon_x+32\varepsilon_y+27\varepsilon_x^2+36\varepsilon_x\varepsilon_y+4\varepsilon_y^2+3\varepsilon_x^3\\
&\qquad+12\varepsilon_x^2\varepsilon_y+4\varepsilon_x\varepsilon_y^2+\varepsilon_x^3\varepsilon_y+\varepsilon_x^2\varepsilon_y^2)
\end{aligned}$$

From this we can extract all of the higher-order partial derivatives.

$$\begin{aligned}
g(x,y)\Big|_{x=2,y=3} &= x^3y+x^2y^2\Big|_{x=2,y=3} = 60 = 0!\times 0!\times 60\\
&= \mathcal{E}\ \varepsilon_x^0\varepsilon_y^0\ (g\ ((2+\varepsilon_x),(3+\varepsilon_y)))\\
\frac{\partial g(x,y)}{\partial x}\Big|_{x=2,y=3} &= 3x^2y+2xy^2\Big|_{x=2,y=3} = 72 = 1!\times 0!\times 72\\
&= \mathcal{E}\ \varepsilon_x^1\varepsilon_y^0\ (g\ ((2+\varepsilon_x),(3+\varepsilon_y)))\\
&\vdots\\
\frac{\partial^4 g(x,y)}{\partial x^3\partial y}\Big|_{x=2,y=3} &= 6\Big|_{x=2,y=3} = 6 = 3!\times 1!\times 1\\
&= \mathcal{E}\ \varepsilon_x^3\varepsilon_y^1\ (g\ ((2+\varepsilon_x),(3+\varepsilon_y)))\\
\frac{\partial^4 g(x,y)}{\partial x^2\partial y^2}\Big|_{x=2,y=3} &= 4\Big|_{x=2,y=3} = 4 = 2!\times 2!\times 1\\
&= \mathcal{E}\ \varepsilon_x^2\varepsilon_y^2\ (g\ ((2+\varepsilon_x),(3+\varepsilon_y)))
\end{aligned}$$

Two difficulties arise when attempting to implement the above. First, there is the need to maintain the distinction between the different εs, addressed in previous work [6]. Second, there is a need to generalize lazy unidimensional streams to the multidimensional case to represent multivariate power series. We address this in the next section.

6. Lazy Multivariate Higher-Order Forward AD

A univariate polynomial $x_0+x_1\varepsilon+x_2\varepsilon^2+\cdots+x_{n-1}\varepsilon^{n-1}+x_n\varepsilon^n$ can be evaluated using Horner's method [9] as

$$x_0+(x_1+(x_2+\cdots+(x_{n-1}+x_n\varepsilon)\varepsilon\cdots)\varepsilon)\varepsilon$$

This indicates that univariate polynomials can be represented as nested dual numbers. Multivariate polynomials can be represented as nested *tagged* dual numbers, i.e., triples of the form $\langle\varepsilon,x,x'\rangle$, with potentially distinct εs, to represent $x+x'\varepsilon$. We assume that there is a total order $\prec$ over the εs and refer to such tagged dual numbers as *linear terms.* Power series can be represented as binary trees whose nodes are linear terms with a lazy x' slot. As illustrated in the code accompanying this paper, we constrain such representations to maintain the following invariants:

I-1 In any linear term $x+x'\varepsilon$, the x slot is either real, or a linear term over ε' where $\varepsilon'\prec\varepsilon$.

I-2 In any linear term $x+x'\varepsilon$, the x' slot is either real, a linear term over ε' where $\varepsilon'\prec\varepsilon$, or a linear term over ε.

These ensure that the coefficient of each term in a multivariate series is stored in at most one leaf. They also simplify proofs of termination of the derivative-taking code and lower the time bound on access to a multivariate power series.

Figure 1 contains a SCHEME implementation of an API for manipulating multivariate power series represented as lazy linear terms. Central to this API are mechanisms $\mathcal{Q}\ \varepsilon\ p$ and $\mathcal{R}\ \varepsilon\ p$ for computing the quotient and remainder when dividing the power series p by the variable ε.

$$\begin{aligned}
\mathcal{R}\ \varepsilon\ r &\triangleq r\ \text{ when } r\in\mathbb{R} \tag{13}\\
\mathcal{R}\ \varepsilon\ (x+x'\varepsilon') &\triangleq x+x'\varepsilon'\ \text{ when } \varepsilon'\prec\varepsilon \tag{14}\\
\mathcal{R}\ \varepsilon\ (x+x'\varepsilon) &\triangleq x \tag{15}\\
\mathcal{R}\ \varepsilon\ (x+x'\varepsilon') &\triangleq (\mathcal{R}\ \varepsilon\ x)+(\mathcal{R}\ \varepsilon\ x')\varepsilon'\ \text{ when } \varepsilon\neq\varepsilon' \tag{16}\\
\mathcal{Q}\ \varepsilon\ r &\triangleq 0\ \text{ when } r\in\mathbb{R} \tag{17}\\
\mathcal{Q}\ \varepsilon\ (x+x'\varepsilon') &\triangleq 0\ \text{ when } \varepsilon'\prec\varepsilon \tag{18}\\
\mathcal{Q}\ \varepsilon\ (x+x'\varepsilon) &\triangleq x' \tag{19}\\
\mathcal{Q}\ \varepsilon\ (x+x'\varepsilon') &\triangleq (\mathcal{Q}\ \varepsilon\ x)+(\mathcal{Q}\ \varepsilon\ x')\varepsilon'\ \text{ when } \varepsilon\neq\varepsilon' \tag{20}
\end{aligned}$$

Cases (14) and (18) and the simplifications in cases (15) and (19) are valid because of the invariants. Note that, unlike an analogous earlier API [6], the above correctly handles linear terms $x+x'\varepsilon$ where x' may itself be a linear term in the same ε. Also note that, because of laziness, unlike that earlier API, there is no easy way to implement the simplification rule $x+0\varepsilon\rightsquigarrow x$.

To perform the nonstandard interpretation, we need to extend the numeric primitives to apply to power series. For simplicity, we do this only for univariate and bivariate primitives. We can assume that the power series arguments to such primitives take the form of a linear term $x+x'\varepsilon$. The extension of a univariate primitive f applied to a linear term $x+x'\varepsilon$ can be derived via a univariate Taylor expansion about x in terms of $x'\varepsilon$.

$$\begin{aligned}
f(x+x'\varepsilon) &= \sum_{i=0}^{\infty}\frac{f^{(i)}(x)}{i!}(x'\varepsilon)^i\\
&= f(x)+\sum_{i=1}^{\infty}\frac{f^{(i)}(x){x'}^i}{i!}\varepsilon^i\\
&= f(x)+\left(\sum_{i=1}^{\infty}\frac{f^{(i)}(x){x'}^{i-1}}{i!}\varepsilon^{i-1}\right)x'\varepsilon\\
&= f(x)+\left(\sum_{i=0}^{\infty}\frac{f^{(i+1)}(x){x'}^i}{(i+1)!}\varepsilon^{i}\right)x'\varepsilon \tag{21}
\end{aligned}$$

In the above, $f^{(i)}$ denotes the i-th derivative of f. Note that

$$f'(x+x'\varepsilon)=\sum_{i=0}^{\infty}\frac{f^{(i+1)}(x){x'}^i}{i!}\varepsilon^i \tag{22}$$

Also note that the right hand side of (22) is similar to the coefficient of $x'\varepsilon$ in (21), differing only in that the derivatives in the power series are divided by $(i+1)!$ instead of $i!$. The coefficient of $x'\varepsilon$ in (21) can be derived from the right hand side of (22) by postulating an operator $\mathcal{C}_{\varepsilon^0}$ to adjust the coefficients.

$$\sum_{i=0}^{\infty}\frac{f^{(i+1)}(x){x'}^i}{(i+1)!}\varepsilon^i=\mathcal{C}_{\varepsilon^0}\ \sum_{i=0}^{\infty}\frac{f^{(i+1)}(x){x'}^i}{i!}\varepsilon^i \tag{23}$$

(As a formal power series operator, $\mathcal{C}_{\varepsilon^0}\ f(\varepsilon)=\frac{1}{\varepsilon}\int_0^{\varepsilon}f(\varepsilon)\,\mathrm{d}\varepsilon$.) This operator can be defined as follows:

$$\begin{aligned}
\mathcal{C}_{\varepsilon^i}\ r &\triangleq \frac{r}{i+1}\ \text{ when } r\in\mathbb{R} \tag{24}\\
\mathcal{C}_{\varepsilon^i}\ (x+x'\varepsilon) &\triangleq (\mathcal{C}_{\varepsilon^i}\ x)+(\mathcal{C}_{\varepsilon^{i+1}}\ x')\varepsilon \tag{25}\\
\mathcal{C}_{\varepsilon^i}\ (x+x'\varepsilon') &\triangleq (\mathcal{C}_{\varepsilon^i}\ x)+(\mathcal{C}_{\varepsilon^i}\ x')\varepsilon'\ \text{ when } \varepsilon\neq\varepsilon' \tag{26}
\end{aligned}$$

```
(define <_e <)

(define =_e =)

(define linear-term? (let ((pair? pair?)) (lambda (p) (and (pair? p) (eq? (car p) ’linear-term)))))

(define-syntax linear-term (syntax-rules () ((linear-term e x x-prime) (list ’linear-term e x (delay x-prime)))))

(define epsilon cadr)

(define (r e p)
 (cond ;; Equation (13)
       ((not (linear-term? p)) p)
       ;; Equation (14)
       ((<_e (epsilon p) e) p)
       ;; Equation (15)
       ((=_e (epsilon p) e) (caddr p))
       ;; Equation (16)
       (else (linear-term (epsilon p) (r e (caddr p)) (r e (force (cadddr p)))))))

(define (q e p)
 (cond ;; Equation (17)
       ((not (linear-term? p)) 0)
       ;; Equation (18)
       ((<_e (epsilon p) e) 0)
       ;; Equation (19)
       ((=_e (epsilon p) e) (force (cadddr p)))
       ;; Equation (20)
       (else (linear-term (epsilon p) (q e (caddr p)) (q e (force (cadddr p)))))))

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

(define (univariate-e e i p)
 (cond ;; Equation (5)
       ((zero? i) (r e p))
       ;; Equation (6)
       (else (* i (univariate-e e (- i 1) (q e p))))))

(define (multivariate-e e i p)
 (cond ;; Equation (10)
       ((null? i) p)
       ;; Equation (11)
       (else (multivariate-e (cdr e) (cdr i) (univariate-e (car e) (car i) p)))))
```

Figure 1. A SCHEME implementation of an API for manipulating multivariate power series represented as lazy linear terms. Note that to support nested invocation of $\mathcal{D}$, the $\times$ in (6) must be the lifted variant that works on power-series arguments. Note that generated εs never escape any equations in which they are generated, i.e., (2, 7, 12, 31–34). Thus one can improve upon the above implementation by allocating and reclaiming εs in a LIFO fashion.

Note that x' in (23) can contain ε. This is a problem because $\mathcal{C}_{\varepsilon^i}$ operates by counting instances of ε, and has no way to distinguish the εs in x' that should not be counted. We solve this by renaming ε to a fresh ε' prior to calling $\mathcal{C}_{\varepsilon^i}$ and renaming ε' back to ε in the result. For the cases that arise in this paper,[2] such renaming can be accomplished with the following:

$$p[\varepsilon \mapsto \varepsilon] \triangleq p \tag{27}$$

$$r[\varepsilon_1 \mapsto \varepsilon_2] \triangleq r \quad \text{when } r \in \mathbb{R} \tag{28}$$

$$(x + x'\varepsilon')[\varepsilon_1 \mapsto \varepsilon_2] \triangleq x + x'\varepsilon' \quad \text{when } \varepsilon' \prec \varepsilon_1 \tag{29}$$

$$(x + x'\varepsilon_1)[\varepsilon_1 \mapsto \varepsilon_2] \triangleq (\mathcal{R}\,\varepsilon_2\,x) + ((\mathcal{Q}\,\varepsilon_2\,x) + x'[\varepsilon_1 \mapsto \varepsilon_2])\varepsilon_2 \tag{30}$$

This yields the following method for extending univariate primitives:

$$f\ (x + x'\varepsilon) \triangleq (f\ x) + ((\mathcal{C}_{\varepsilon^0}\ (f'\ (x + x'\varepsilon)[\varepsilon \mapsto \varepsilon']))[\varepsilon' \mapsto \varepsilon] \times x')\varepsilon \tag{31}$$

This requires supplying f', the first derivative of each univariate primitive f.

Bivariate primitives can be extended when the first argument is a linear term over ε and the second argument is either a real or a linear term over ε' where $\varepsilon' \prec \varepsilon$ by performing a univariate Taylor expansion around the first argument.

$$f\ ((x + x'\varepsilon), y) \triangleq (f\ (x, y)) + ((\mathcal{C}_{\varepsilon^0}\ (f_1\ ((x + x'\varepsilon)[\varepsilon \mapsto \varepsilon'], y)))[\varepsilon' \mapsto \varepsilon] \times x')\varepsilon \tag{32}$$

Analogously, bivariate primitives can be extended when the second argument is a linear term over ε and the first argument is either a real or a linear term over ε' where $\varepsilon' \prec \varepsilon$ by performing a univariate Taylor expansion around the second argument.

$$f\ (x, (y + y'\varepsilon)) \triangleq (f\ (x, y)) + ((\mathcal{C}_{\varepsilon^0}\ (f_2\ (x, (y + y'\varepsilon)[\varepsilon \mapsto \varepsilon'])))[\varepsilon' \mapsto \varepsilon] \times y')\varepsilon \tag{33}$$

This requires supplying f_1 and f_2, the partial first derivatives of each bivariate primitive f with respect to its first and second arguments respectively.

To handle the case when both arguments are linear terms over the same ε we rename the ε in one argument to a fresh ε', reducing this case to either (32) or (33), and then rename ε' back to ε in the result.

$$f\ ((x + x'\varepsilon), (y + y'\varepsilon)) \triangleq (f\ ((x + x'\varepsilon), (y + y'\varepsilon)[\varepsilon \mapsto \varepsilon']))[\varepsilon' \mapsto \varepsilon] \tag{34}$$

These techniques are implemented in figure 2 and used to overload some SCHEME primitives in figure 3. Figure 4 completes the implementation. Note that the computational efficiency of this implementation relies on the fact that standard SCHEME memoizes the results of forcing promises. The code from figures 1–4 is available from `http://www.bcl.hamilton.ie/~qobi/tower/`.

[2] Renaming is only applied in equations (31–34) and therein only in cases where $\varepsilon \prec \varepsilon'$ for any existing ε. Furthermore, (29) is valid only because of the invariants.

```
(define (c e i p)
 (cond ;; Equation (24)
       ((not (linear-term? p)) (/ p (+ i 1)))
       ;; Equation (25)
       ((=_e (epsilon p) e) (linear-term (epsilon p) (c e i (r (epsilon p) p)) (c e (+ i 1) (q (epsilon p) p))))
       ;; Equation (26)
       (else (linear-term (epsilon p) (c e i (r (epsilon p) p)) (c e i (q (epsilon p) p))))))

(define (rename e1 e2 p)
 (cond ;; Equation (27)
       ((=_e e1 e2) p)
       ;; Equation (28)
       ((not (linear-term? p)) p)
       ;; Equation (29)
       ((<_e (epsilon p) e1) p)
       ;; Equation (30)
       ((=_e (epsilon p) e1) (linear-term e2 (r e2 (r e1 p)) (+ (q e2 (r e1 p)) (rename e1 e2 (q e1 p)))))
       (else (error "This case should never occur in this program."))))

(define (lift-real->real f df/dx)
 (letrec ((self (lambda (p)
                 (cond ;; Equation (31)
                       ((linear-term? p)
                        (let ((e (epsilon p)))
                         (linear-term e
                                      (self (r e p))
                                      (* (let ((e-prime (generate-epsilon)))
                                          (rename e-prime e (c e-prime 0 (df/dx (linear-term e-prime (r e p) (q e p))))))
                                         (q e p)))))
                       (else (f p))))))
  self))

(define (lift-real*real->real f df/dx1 df/dx2)
 (letrec ((self
           (lambda (p1 p2)
            (cond
             ;; Equation (32)
             ((and (linear-term? p1) (or (not (linear-term? p2)) (<_e (epsilon p2) (epsilon p1))))
              (let ((e1 (epsilon p1)))
               (linear-term e1
                            (self (r e1 p1) p2)
                            (* (let ((e-prime (generate-epsilon)))
                                (rename e-prime e1 (c e-prime 0 (df/dx1 (linear-term e-prime (r e1 p1) (q e1 p1)) p2))))
                               (q e1 p1)))))
             ;; Equation (33)
             ((and (linear-term? p2) (or (not (linear-term? p1)) (<_e (epsilon p1) (epsilon p2))))
              (let ((e2 (epsilon p2)))
               (linear-term e2
                            (self p1 (r e2 p2))
                            (* (let ((e-prime (generate-epsilon)))
                                (rename e-prime e2 (c e-prime 0 (df/dx2 p1 (linear-term e-prime (r e2 p2) (q e2 p2))))))
                               (q e2 p2)))))
             ;; Equation (34)
             ((and (linear-term? p1) (linear-term? p2) (=_e (epsilon p1) (epsilon p2)))
              (let ((e (epsilon p1)) (e-prime (generate-epsilon))) (rename e-prime e (self p1 (rename e e-prime p2)))))
             (else (f p1 p2))))))
  self))

(define (r* p) (if (linear-term? p) (r* (r (epsilon p) p)) p))

(define (lift-real\symbol{94}n->boolean f) (lambda ps (apply f (map r* ps))))
```

Figure 2. A mechanism for extending SCHEME procedures of type $\mathbb{R} \to \mathbb{R}$, $\mathbb{R} \times \mathbb{R} \to \mathbb{R}$, and $\mathbb{R}^n \to$ **boolean** to support multivariate power series. Note that the $+$ in (30) and the $\times$ in (31–33) must be the lifted variant that works on power-series arguments. Furthermore, f' in (31), f_1 in (32), and f_2 in (33) must internally use the lifted variants of operations that work on power-series arguments.

```
(define pair? (let ((pair? pair?)) (lambda (x) (and (pair? x) (not (linear-term? 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?))
```

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

```
;;; Equation (2)
(define (derivative f) (lambda (c) (let ((e (generate-epsilon))) (univariate-e e 1 (f (linear-term e c 1))))))

(define (ith-derivative-by-repetition i f)
 (cond ;; Equation (3)
       ((zero? i) f)
       ;; Equation (4)
       (else (ith-derivative-by-repetition (- i 1) (derivative f)))))

;;; Equation (7)
(define (ith-derivative-by-tower i f)
 (lambda (c) (let ((e (generate-epsilon))) (univariate-e e i (f (linear-term e c 1))))))

(define (position-of-nonzero i)
 (cond ((null? i) #f)
       ((zero? (car i)) (let ((position (position-of-nonzero (cdr i)))) (if position (+ position 1) #f)))
       (else 0)))

(define (decrement-lth i l) (if (zero? l) (cons (- (car i) 1) (cdr i)) (cons (car i) (decrement-lth (cdr i) (- l 1)))))

(define (list-replace-lth c l u) (if (zero? l) (cons u (cdr c)) (cons (car c) (list-replace-lth (cdr c) (- l 1) u))))

(define (partial-derivative-by-repetition i f)
 (let ((l (position-of-nonzero i)))
  (cond ;; Equation (8)
        ((not l) f)
        ;; Equation (9)
        (else (partial-derivative-by-repetition
               (decrement-lth i l) (lambda (c) ((derivative (lambda (u) (f (list-replace-lth c l u)))) (list-ref c l))))))))

;;; Equation (12)
(define (partial-derivative-by-tower i f)
 (lambda (c)
  (let ((e (map (lambda (cl) (generate-epsilon)) c)))
   (multivariate-e e i (f (map (lambda (el cl) (linear-term el cl 1)) e c))))))
```

Figure 4. A SCHEME implementation of $\mathcal{D}$, the repetition and tower methods for $\mathcal{D}^i$, and the repetition and tower methods for $\mathcal{D}^{(i_1,\ldots,i_n)}$.

7. Discussion

Forward AD is typically formulated using very different machinery than that used above. The univariate first-order case is usually formulated as a transformation of a program whose program points compute values $f(x)$ of the program input x to one whose program points compute values $\langle f(x), f'(x)\rangle$. Since the program is a composition of primitives, the program transformation, as well as the transformation of the primitives, are formulated as applications of the chain rule. Karczmarczuk formulated the univariate higher-order case as a similar transformation to a program whose program points compute stream values $(f(x), f'(x), f''(x), \ldots)$ via the chain rule, though he did not present a derivation of the transformations of the primitives.

The streams we have used are similar, but contain factorial factors: $(f(x)/0!, f'(x)/1!, f''(x)/2!, \ldots, f^{(i)}(x)/i!, \ldots)$. These Taylor series streams simplify some bookkeeping, in particular allowing the use of Taylor expansions instead of the chain rule in the derivations. This makes the multivariate higher-order case more straightforward to derive and justify. However, since each representation can be converted to the other (using operators that are similar to $\mathcal{C}$), we do not consider this a fundamental difference.

Karczmarczuk [4] hinted at a formulation of the multivariate higher-order case using the chain rule, where lazy unidimensional streams are replaced with lazy trees, but did not present a derivation or justification of the method's correctness. That method redundantly represents identical cross derivatives, i.e., $\partial^2 f/\partial x_i \partial x_j = \partial^2 f/\partial x_j \partial x_i$. Our method avoids that inefficiency. Moreover, although nesting is not our topic, the code presented does allow the derivative-taking constructs to nest correctly.

Laziness is particularly useful when representing and manipulating power series, in contexts beyond those considered here [10]. For instance it can be used to define a power series with a recurrence relation. Such power series arise naturally in related contexts, such as differential equations that cannot be solved in closed form. Formulating nestable multivariate higher-order forward AD in terms of lazy power-series representations can allow forward AD to inter-operate with such other applications of power series.

Acknowledgments

This work was supported, in part, by NSF grant CCF-0438806, Science Foundation Ireland grant 00/PI.1/C067, and a grant from the Higher Education Authority of Ireland. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

[1] R. E. Wengert, "A simple automatic derivative evaluation program," *Comm. of the ACM*, vol. 7, no. 8, pp. 463–4, 1964.

[2] J. Karczmarczuk, "Functional differentiation of computer programs," in *Proceedings of the III ACM SIGPLAN International Conference on Functional Programming*, Baltimore, MD, Sept. 1998, pp. 195–203.

[3] ——, "Lazy differential algebra and its applications," in *Workshop, III International Summer School on Advanced Functional Programming*, Braga, Portugal, Sept. 1998.

[4] ——, "Functional coding of differential forms," in *Scottish Workshop on FP*, Sept. 1999.

[5] ——, "Functional differentiation of computer programs," *Journal of Higher-Order and Symbolic Computation*, vol. 14, pp. 35–57, 2001.

[6] J. M. Siskind and B. A. Pearlmutter, "Nesting forward-mode AD in a functional framework," to appear.

[7] B. Taylor, *Methodus Incrementorum Directa et Inversa*. London, 1715.

[8] W. K. Clifford, "Preliminary sketch of bi-quaternions," *Proceedings of the London Mathematical Society*, vol. 4, pp. 381–395, 1873.

[9] W. G. Horner, "A new method of solving numerical equations of all orders, by continuous approximation," *Philos. Trans. Roy. Soc. London*, vol. 109, pp. 308–335, July 1819.

[10] J. Karczmarczuk, "Generating power of lazy semantics," *Theoretical Computer Science*, vol. 187, 1997.