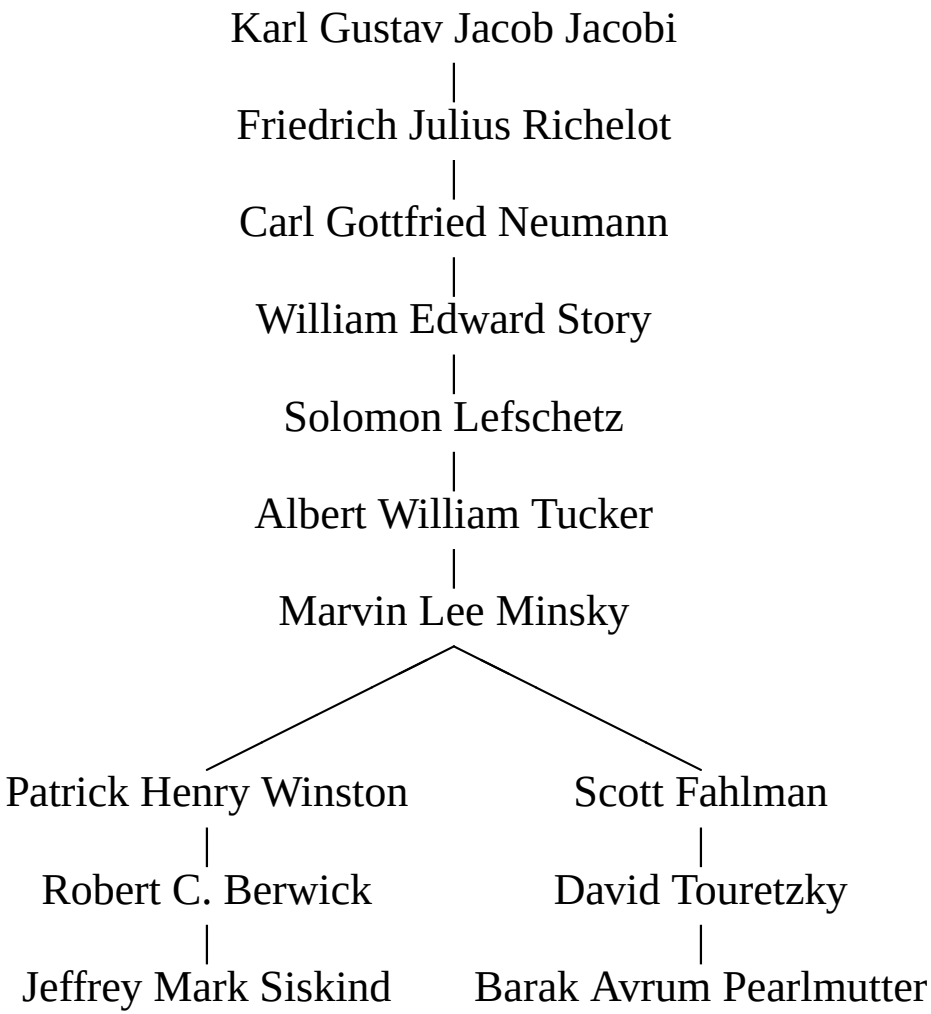
Automatic Differentiation of Functional Programs or Lambda the Ultimate Calculus or How I Learned to Stop Worrying and Love Map-Closure

Jeffrey Mark Siskind
qobi@purdue.edu

School of Electrical and Computer Engineering
Purdue University

Indiana University
20 February 2007

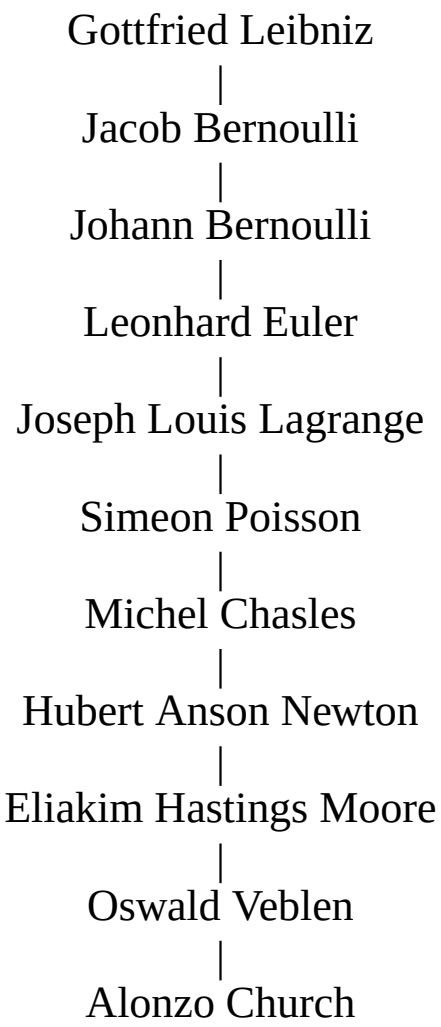
Joint work with Barak A. Pearlmutter.


Karl Gustav Jacob Jacobi
Friedrich Julius Richelot
Carl Gottfried Neumann
William Edward Story
Solomon Lefschetz
Albert William Tucker
Marvin Lee Minsky
Patrick Henry Winston
Scott Fahlman
Robert C. Berwick
David Touretzky
Jeffrey Mark Siskind
Barak Avrum Pearlmutter

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.

(¶4)

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


Gottfried Leibniz
Jacob Bernoulli
Johann Bernoulli
Leonhard Euler
Joseph Louis Lagrange
Simeon Poisson
Michel Chasles
Hubert Anson Newton
Eliakim Hastings Moore
Oswald Veblen
Alonzo Church

Leibnitz (1664) + Church (1941) = Siskind & Pearlmutter (2007)

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*.

Everything You Always Wanted to Know About the Lambda Calculus*

*But Were Afraid To Ask

Everything You Always Wanted to Know About the Lambda Calculus*

(in 7 slides)

*But Were Afraid To Ask

Everything You Always Wanted to Know About the Lambda Calculus*

(in 7 slides)

*But Were Afraid To Ask
(omitted for this audience)

Differential Calculus for Dummies

Differential Calculus for Dummies

(in 6 slides)

Derivatives

$$\frac{\mathrm{d}ax^2}{\mathrm{d}x} \rightsquigarrow 2ax$$

Derivatives

$$\frac{\mathrm{d}ax^2}{\mathrm{d}x} \rightsquigarrow 2ax$$

$$\frac{\mathrm{d}}{\mathrm{d}x} : \underbrace{f}_{\mathbb{R}\to\mathbb{R}} \mapsto \underbrace{f'}_{\mathbb{R}\to\mathbb{R}}$$

Derivatives

$$\frac{\mathrm{d}ax^2}{\mathrm{d}x} \rightsquigarrow 2ax$$

$$\frac{\mathrm{d}}{\mathrm{d}x} : \underbrace{f}_{\mathbb{R}\to\mathbb{R}} \mapsto \underbrace{f'}_{\mathbb{R}\to\mathbb{R}}$$

$$\frac{\mathrm{d}}{\mathrm{d}x} : (\mathbb{R} \to \mathbb{R}) \to (\mathbb{R} \to \mathbb{R})$$

Derivatives

$$\frac{\mathrm{d}ax^2}{\mathrm{d}x} \rightsquigarrow 2ax$$

$$\frac{\mathrm{d}}{\mathrm{d}x} : \underbrace{f}_{\mathbb{R}\to\mathbb{R}} \mapsto \underbrace{f'}_{\mathbb{R}\to\mathbb{R}}$$

$$\frac{\mathrm{d}}{\mathrm{d}x} : (\mathbb{R} \to \mathbb{R}) \to (\mathbb{R} \to \mathbb{R})$$

$$\mathcal{D} : (\mathbb{R} \to \mathbb{R}) \to (\mathbb{R} \to \mathbb{R})$$

Derivatives

$$\frac{\mathrm{d}ax^2}{\mathrm{d}x} \rightsquigarrow 2ax$$

$$\frac{\mathrm{d}}{\mathrm{d}x} : \underbrace{f}_{\mathbb{R}\to\mathbb{R}} \mapsto \underbrace{f'}_{\mathbb{R}\to\mathbb{R}}$$

$$\frac{\mathrm{d}}{\mathrm{d}x} : (\mathbb{R} \to \mathbb{R}) \to (\mathbb{R} \to \mathbb{R})$$

$$\mathcal{D} : (\mathbb{R} \to \mathbb{R}) \to (\mathbb{R} \to \mathbb{R})$$

$$\mathcal{D}\ \lambda x\ ax^2$$

Partial Derivatives

$$\frac{\partial a x^2 y^3}{\partial x} \qquad \frac{\partial a x^2 y^3}{\partial y}$$

Partial Derivatives

$$\frac{\partial a x^2 y^3}{\partial x} \qquad \frac{\partial a x^2 y^3}{\partial y}$$

$$\mathcal{D}\ \lambda x\ a x^2 y^3 \qquad \mathcal{D}\ \lambda y\ a x^2 y^3$$

Partial Derivatives

$$\frac{\partial a x^2 y^3}{\partial x} \qquad \frac{\partial a x^2 y^3}{\partial y}$$

$$\mathcal{D}\ \lambda x\ a x^2 y^3 \qquad \mathcal{D}\ \lambda y\ a x^2 y^3$$

$$\mathcal{D}_1\ \lambda(x,y)\ a x^2 y^3 \qquad \mathcal{D}_2\ \lambda(x,y)\ a x^2 y^3$$

Partial Derivatives

$$\frac{\partial a x^2 y^3}{\partial x} \qquad \frac{\partial a x^2 y^3}{\partial y}$$

$$\mathcal{D}\ \lambda x\ a x^2 y^3 \qquad \mathcal{D}\ \lambda y\ a x^2 y^3$$

$$\mathcal{D}_1\ \lambda(x,y)\ a x^2 y^3 \qquad \mathcal{D}_2\ \lambda(x,y)\ a x^2 y^3$$

$$\frac{\partial}{\partial x} : \underbrace{f}_{\mathbb{R}^n \to \mathbb{R}} \mapsto \underbrace{f'}_{\mathbb{R}^n \to \mathbb{R}}$$

Partial Derivatives

$$\frac{\partial a x^2 y^3}{\partial x} \qquad \frac{\partial a x^2 y^3}{\partial y}$$

$$\mathcal{D}\ \lambda x\ a x^2 y^3 \qquad \mathcal{D}\ \lambda y\ a x^2 y^3$$

$$\mathcal{D}_1\ \lambda(x,y)\ a x^2 y^3 \qquad \mathcal{D}_2\ \lambda(x,y)\ a x^2 y^3$$

$$\frac{\partial}{\partial x} : \underbrace{f}_{\mathbb{R}^n \to \mathbb{R}} \mapsto \underbrace{f'}_{\mathbb{R}^n \to \mathbb{R}}$$

$$\frac{\partial}{\partial x} : (\mathbb{R}^n \to \mathbb{R}) \to (\mathbb{R}^n \to \mathbb{R})$$

Partial Derivatives

$$\frac{\partial a x^2 y^3}{\partial x} \qquad \frac{\partial a x^2 y^3}{\partial y}$$

$$\mathcal{D}\ \lambda x\ a x^2 y^3 \qquad \mathcal{D}\ \lambda y\ a x^2 y^3$$

$$\mathcal{D}_1\ \lambda(x,y)\ a x^2 y^3 \qquad \mathcal{D}_2\ \lambda(x,y)\ a x^2 y^3$$

$$\frac{\partial}{\partial x} : \underbrace{f}_{\mathbb{R}^n \to \mathbb{R}} \mapsto \underbrace{f'}_{\mathbb{R}^n \to \mathbb{R}}$$

$$\frac{\partial}{\partial x} : (\mathbb{R}^n \to \mathbb{R}) \to (\mathbb{R}^n \to \mathbb{R})$$

$$\mathcal{D}_i : (\mathbb{R}^n \to \mathbb{R}) \to (\mathbb{R}^n \to \mathbb{R})$$

Gradients

$$\nabla f\ \mathbf{x} \quad = \quad (\mathcal{D}_1 f\ \mathbf{x}), \ldots, (\mathcal{D}_n f\ \mathbf{x})$$

$$\nabla \quad : \quad (\mathbb{R}^n \to \mathbb{R}) \to (\mathbb{R}^n \to \mathbb{R}^n)$$

Jacobians

$$f : \mathbb{R}^n \to \mathbb{R}^m$$

$$\mathbf{f} : (\mathbb{R}^n \to \mathbb{R})^m$$

$$(\mathcal{J}\ f\ \mathbf{x})[i,j] = (\nabla\ (\mathbf{f}[i]))[j]$$

$$\mathcal{J} : (\mathbb{R}^n \to \mathbb{R}^m) \to (\mathbb{R}^n \to \mathbb{R}^{m \times n})$$

Operators

$\mathcal{D}$, ∇, and $\mathcal{J}$ are traditionally called *operators*.
A more modern term is *higher-order functions*.
Higher-order functions are common in mathematics, physics, and engineering:

> *summations*, *comprehensions*, *quantifications*, *optimizations*, *integrals*, *convolutions*, *filters*, *edge detectors*, *Fourier transforms*, *differential equations*, *Hamiltonians*, . . .

The Chain Rule

$$(f \circ g)\ x = g\ (f\ x)$$

The Chain Rule

$$(f \circ g)\ x = g\ (f\ x)$$

$$\frac{\mathrm{d}g}{\mathrm{d}x} = \frac{\mathrm{d}g}{\mathrm{d}f}\frac{\mathrm{d}f}{\mathrm{d}x}$$

The Chain Rule

$$(f \circ g)\ x = g\ (f\ x)$$

$$\frac{\mathrm{d}g}{\mathrm{d}x} = \frac{\mathrm{d}g}{\mathrm{d}f}\frac{\mathrm{d}f}{\mathrm{d}x}$$

$$\mathcal{D}\ (f \circ g)\ x = (\mathcal{D}\ g\ (f\ x)) \times (\mathcal{D}\ f\ x)$$

The Chain Rule

$$(f \circ g)\ x = g\ (f\ x)$$

$$\frac{\mathrm{d}g}{\mathrm{d}x} = \frac{\mathrm{d}g}{\mathrm{d}f}\frac{\mathrm{d}f}{\mathrm{d}x}$$

$$\mathcal{D}\ (f \circ g)\ x = (\mathcal{D}\ g\ (f\ x)) \times (\mathcal{D}\ f\ x)$$

$$\mathcal{J}\ (f \circ g)\ \mathbf{x} = (\mathcal{J}\ g\ (f\ \mathbf{x})) \times (\mathcal{J}\ f\ \mathbf{x})$$

The Essence of Forward-Mode AD

Taylor expansion:

$$f(c+\varepsilon) = \frac{f(c)}{0!} + \frac{f'(c)}{1!}\varepsilon + \frac{f''(c)}{2!}\varepsilon^2 + \cdots + \frac{f^{(i)}(c)}{i!}\varepsilon^i + \cdots$$

The Essence of Forward-Mode AD

Taylor expansion:

$$f(c+\varepsilon) = \frac{f(c)}{0!} + \frac{f'(c)}{1!}\varepsilon + \frac{f''(c)}{2!}\varepsilon^2 + \cdots + \frac{f^{(i)}(c)}{i!}\varepsilon^i + \cdots$$

To compute $\mathcal{D}\, f\ c$:

The Essence of Forward-Mode AD

Taylor expansion:

$$f(c+\varepsilon) = \frac{f(c)}{0!} + \frac{f'(c)}{1!}\varepsilon + \frac{f''(c)}{2!}\varepsilon^2 + \cdots + \frac{f^{(i)}(c)}{i!}\varepsilon^i + \cdots$$

To compute $\mathcal{D}\, f\ c$:

- evaluate f

The Essence of Forward-Mode AD

Taylor expansion:

$$f(c+\varepsilon) = \frac{f(c)}{0!} + \frac{f'(c)}{1!}\varepsilon + \frac{f''(c)}{2!}\varepsilon^2 + \cdots + \frac{f^{(i)}(c)}{i!}\varepsilon^i + \cdots$$

To compute $\mathcal{D}\, f\ c$:

- evaluate f at the **term** $c + \varepsilon$

The Essence of Forward-Mode AD

Taylor expansion:

$$f(c+\varepsilon) = \frac{f(c)}{0!} + \frac{f'(c)}{1!}\varepsilon + \frac{f''(c)}{2!}\varepsilon^2 + \cdots + \frac{f^{(i)}(c)}{i!}\varepsilon^i + \cdots$$

To compute $\mathcal{D}\ f\ c$:

- evaluate f at the **term** $c + \varepsilon$ to get a **power series**,

The Essence of Forward-Mode AD

Taylor expansion:

$$f(c+\varepsilon) = \frac{f(c)}{0!} + \frac{f'(c)}{1!}\varepsilon + \frac{f''(c)}{2!}\varepsilon^2 + \cdots + \frac{f^{(i)}(c)}{i!}\varepsilon^i + \cdots$$

To compute $\mathcal{D}\ f\ c$:

- evaluate f at the **term** $c + \varepsilon$ to get a **power series**,
- extract the coefficient of ε,

The Essence of Forward-Mode AD

Taylor expansion:

$$f(c+\varepsilon) = \frac{f(c)}{0!} + \frac{f'(c)}{1!}\varepsilon + \frac{f''(c)}{2!}\varepsilon^2 + \cdots + \frac{f^{(i)}(c)}{i!}\varepsilon^i + \cdots$$

To compute $\mathcal{D}\ f\ c$:

- evaluate f at the **term** $c + \varepsilon$ to get a **power series**,
- extract the coefficient of ε,

The Essence of Forward-Mode AD

Taylor expansion:

$$f(c+\varepsilon) = \frac{f(c)}{0!} + \frac{f'(c)}{1!}\varepsilon + \frac{f''(c)}{2!}\varepsilon^2 + \cdots + \frac{f^{(i)}(c)}{i!}\varepsilon^i + \cdots$$

To compute $\mathcal{D}\ f\ c$:

- evaluate f at the **term** $c + \varepsilon$ to get a **power series**,
- extract the coefficient of ε, and
- multiply by 1!

The Essence of Forward-Mode AD

Taylor expansion:

$$f(c+\varepsilon) = \frac{f(c)}{0!} + \frac{f'(c)}{1!}\varepsilon + \frac{f''(c)}{2!}\varepsilon^2 + \cdots + \frac{f^{(i)}(c)}{i!}\varepsilon^i + \cdots$$

To compute $\mathcal{D}\, f\, c$:

- evaluate f at the **term** $c + \varepsilon$ to get a **power series**,
- extract the coefficient of ε, and
- multiply by 1! (noop).

The Essence of Forward-Mode AD

Taylor expansion:

$$f(c+\varepsilon) = \frac{f(c)}{0!} + \frac{f'(c)}{1!}\varepsilon + \frac{f''(c)}{2!}\varepsilon^2 + \cdots + \frac{f^{(i)}(c)}{i!}\varepsilon^i + \cdots$$

To compute $\mathcal{D}\, f\, c$:

- evaluate f at the **term** $c + \varepsilon$ to get a **power series**,
- extract the coefficient of ε, and
- multiply by 1! (noop).

Key idea: Only need output to be a **finite** truncated power series $a + b\varepsilon$.

The Essence of Forward-Mode AD

Taylor expansion:

$$f(c+\varepsilon) = \frac{f(c)}{0!} + \frac{f'(c)}{1!}\varepsilon + \frac{f''(c)}{2!}\varepsilon^2 + \cdots + \frac{f^{(i)}(c)}{i!}\varepsilon^i + \cdots$$

To compute $\mathcal{D}\, f\, c$:

- evaluate f at the **term** $c + \varepsilon$ to get a **power series**,
- extract the coefficient of ε, and
- multiply by 1! (noop).

Key idea: Only need output to be a **finite** truncated power series $a + b\varepsilon$.

The input $c + \varepsilon$ is also a truncated power series.

The Essence of Forward-Mode AD

Taylor expansion:

$$f(c+\varepsilon) = \frac{f(c)}{0!} + \frac{f'(c)}{1!}\varepsilon + \frac{f''(c)}{2!}\varepsilon^2 + \cdots + \frac{f^{(i)}(c)}{i!}\varepsilon^i + \cdots$$

To compute $\mathcal{D}\ f\ c$:

- evaluate f at the **term** $c + \varepsilon$ to get a **power series**,
- extract the coefficient of ε, and
- multiply by 1! (noop).

Key idea: Only need output to be a **finite** truncated power series $a + b\varepsilon$.

The input $c + \varepsilon$ is also a truncated power series.

Can do a *nonstandard interpretation* of f over truncated power series.

The Essence of Forward-Mode AD

Taylor expansion:

$$f(c+\varepsilon) = \frac{f(c)}{0!} + \frac{f'(c)}{1!}\varepsilon + \frac{f''(c)}{2!}\varepsilon^2 + \cdots + \frac{f^{(i)}(c)}{i!}\varepsilon^i + \cdots$$

To compute $\mathcal{D}\, f\, c$:

- evaluate f at the **term** $c + \varepsilon$ to get a **power series**,
- extract the coefficient of ε, and
- multiply by 1! (noop).

Key idea: Only need output to be a **finite** truncated power series $a + b\varepsilon$.

The input $c + \varepsilon$ is also a truncated power series.

Can do a *nonstandard interpretation* of f over truncated power series.

Preserves control flow: Augments original values with derivatives.

The Essence of Forward-Mode AD

Taylor expansion:

$$f(c+\varepsilon) = \frac{f(c)}{0!} + \frac{f'(c)}{1!}\varepsilon + \frac{f''(c)}{2!}\varepsilon^2 + \cdots + \frac{f^{(i)}(c)}{i!}\varepsilon^i + \cdots$$

To compute $\mathcal{D}\ f\ c$:

- evaluate f at the **term** $c + \varepsilon$ to get a **power series**,
- extract the coefficient of ε, and
- multiply by 1! (noop).

Key idea: Only need output to be a **finite** truncated power series $a + b\varepsilon$.

The input $c + \varepsilon$ is also a truncated power series.

Can do a *nonstandard interpretation* of f over truncated power series.

Preserves control flow: Augments original values with derivatives.

$(\mathcal{D}\ f)$ is $\mathcal{O}(1)$ relative to f (both space and time).

The Essence of Forward-Mode AD

Taylor expansion:

$$f(c+\varepsilon) = \frac{f(c)}{0!} + \frac{f'(c)}{1!}\varepsilon + \frac{f''(c)}{2!}\varepsilon^2 + \cdots + \frac{f^{(i)}(c)}{i!}\varepsilon^i + \cdots$$

To compute $\mathcal{D}\ f\ c$:

- evaluate f at the **term** $c + \varepsilon$ to get a **power series**,
- extract the coefficient of ε, and
- multiply by 1! (noop).

Key idea: Only need output to be a **finite** truncated power series $a + b\varepsilon$.

The input $c + \varepsilon$ is also a truncated power series.

Can do a *nonstandard interpretation* of f over truncated power series.

Preserves control flow: Augments original values with derivatives.

$(\mathcal{D}\ f)$ is $\mathcal{O}(1)$ relative to f (both space and time).

These $a + b\varepsilon$ are called *dual numbers* and can be represented as $\langle a, b \rangle$.

The Essence of Forward-Mode AD

Taylor expansion:

$$f(c+\varepsilon) = \frac{f(c)}{0!} + \frac{f'(c)}{1!}\varepsilon + \frac{f''(c)}{2!}\varepsilon^2 + \cdots + \frac{f^{(i)}(c)}{i!}\varepsilon^i + \cdots$$

To compute $\mathcal{D}\ f\ c$:

- evaluate f at the **term** $c + \varepsilon$ to get a **power series**,
- extract the coefficient of ε, and
- multiply by 1! (noop).

Key idea: Only need output to be a **finite** truncated power series $a + b\varepsilon$.

The input $c + \varepsilon$ is also a truncated power series.

Can do a *nonstandard interpretation* of f over truncated power series.

Preserves control flow: Augments original values with derivatives.

$(\mathcal{D}\ f)$ is $\mathcal{O}(1)$ relative to f (both space and time).

These $a + b\varepsilon$ are called *dual numbers* and can be represented as $\langle a, b \rangle$.

(Analogous to complex numbers $a + b\mathrm{i}$ represented as $\langle a, b \rangle$.)

Taylor Expansions and Dual Numbers

Taylor, B. (1715). *Methodus Incrementorum Directa et Inversa*, London.

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

Arithmetic on Truncated Power Series (i.e. Dual Numbers)

$$(x_0 + x_1\varepsilon + \mathcal{O}(\varepsilon^2)) + (y_0 + y_1\varepsilon + \mathcal{O}(\varepsilon^2)) = (x_0 + y_0) + (x_1 + y_1)\varepsilon + \mathcal{O}(\varepsilon^2)$$

Arithmetic on Truncated Power Series (i.e. Dual Numbers)

$$(x_0 + x_1\varepsilon + \mathcal{O}(\varepsilon^2)) + (y_0 + y_1\varepsilon + \mathcal{O}(\varepsilon^2)) = (x_0 + y_0) + (x_1 + y_1)\varepsilon + \mathcal{O}(\varepsilon^2)$$

$$\begin{aligned}(x_0 + x_1\varepsilon + \mathcal{O}(\varepsilon^2)) \times (y_0 + y_1\varepsilon + \mathcal{O}(\varepsilon^2)) \\ = (x_0 \times y_0) + (x_0 \times y_1 + x_1 \times y_0)\varepsilon + \mathcal{O}(\varepsilon^2)\end{aligned}$$

Arithmetic on Truncated Power Series (i.e. Dual Numbers)

$$(x_0 + x_1\varepsilon + \mathcal{O}(\varepsilon^2)) + (y_0 + y_1\varepsilon + \mathcal{O}(\varepsilon^2)) = (x_0 + y_0) + (x_1 + y_1)\varepsilon + \mathcal{O}(\varepsilon^2)$$

$$\begin{aligned}(x_0 + x_1\varepsilon + \mathcal{O}(\varepsilon^2)) &\times (y_0 + y_1\varepsilon + \mathcal{O}(\varepsilon^2)) \\ &= (x_0 \times y_0) + (x_0 \times y_1 + x_1 \times y_0)\varepsilon + \mathcal{O}(\varepsilon^2)\end{aligned}$$

$$u\ (x_0 + x_1\varepsilon + \mathcal{O}(\varepsilon^2)) = (u\ x_0) + (x_1 \times (u'\ x_0))\varepsilon + \mathcal{O}(\varepsilon^2)$$

Arithmetic on Truncated Power Series (i.e. Dual Numbers)

$$(x_0 + x_1\varepsilon + \mathcal{O}(\varepsilon^2)) + (y_0 + y_1\varepsilon + \mathcal{O}(\varepsilon^2)) = (x_0 + y_0) + (x_1 + y_1)\varepsilon + \mathcal{O}(\varepsilon^2)$$

$$\begin{aligned}(x_0 + x_1\varepsilon + \mathcal{O}(\varepsilon^2)) \times (y_0 + y_1\varepsilon + \mathcal{O}(\varepsilon^2)) \\ = (x_0 \times y_0) + (x_0 \times y_1 + x_1 \times y_0)\varepsilon + \mathcal{O}(\varepsilon^2)\end{aligned}$$

$$u\ (x_0 + x_1\varepsilon + \mathcal{O}(\varepsilon^2)) = (u\ x_0) + (x_1 \times (u'\ x_0))\varepsilon + \mathcal{O}(\varepsilon^2)$$

$$\begin{aligned}b\ ((x_0 + x_1\varepsilon + \mathcal{O}(\varepsilon^2)), (y_0 + y_1\varepsilon + \mathcal{O}(\varepsilon^2))) \\ = (b\ (x_0, y_0)) + (x_1 \times (b^{(1,0)}\ (x_0, y_0)) + y_1 \times (b^{(0,1)}\ (x_0, y_0)))\varepsilon + \mathcal{O}(\varepsilon^2)\end{aligned}$$

Traditional Forward-Mode AD

$$\mathbb{R}^n \to \mathbb{R}^m \quad \rightsquigarrow \quad (\mathbb{R}^n \times \mathbb{R}^n) \to (\mathbb{R}^m \times \mathbb{R}^m)$$
$$\mathbb{R}^n \to \mathbb{R}^m \quad \rightsquigarrow \quad (\mathbb{R} \times \mathbb{R})^n \to (\mathbb{R} \times \mathbb{R})^m$$

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

In differential algebra, dual numbers are known as *bundles* of (primal) values x and their *tangents* $\overline{x'}$.

$$x \rhd \overline{x'}$$

```
(define-structure bundle primal tangent)
```

Tangent Spaces

$$
\begin{aligned}
\overline{v'} &\in \mathbb{R} \quad \text{when } v \in \mathbb{R} \\
\overline{v'} &= v \quad \text{when } v \text{ is a discrete scalar} \\
\overline{v \rhd \overline{v'}} &= \overline{v'} \rhd \overline{\overline{v'}} \\
\overline{v_1, v_2'} &= \overline{v_1'}, \overline{v_2'} \\
\overline{\langle \{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\}, \lambda x\ e \rangle} &= \langle \{x_1 \mapsto \overline{v_1'}, \ldots, x_n \mapsto \overline{v_n'}\}, \lambda x\ e \rangle
\end{aligned}
$$

$$v \blacktriangleright \overline{v'} \stackrel{\triangle}{=} v \rhd \overline{v'} \qquad \text{when } v \text{ is a scalar}$$

$$v \blacktriangleright \overline{v'} \stackrel{\triangle}{=} \overrightarrow{v} \qquad \text{when } v \text{ is a primitive}$$

$$(v \rhd \overline{v'}) \blacktriangleright \overline{(v \rhd \overline{v'})} \stackrel{\triangle}{=} (v \blacktriangleright \overline{v'}) \rhd (\overline{v'} \blacktriangleright \overline{\overline{v'}})$$

$$(v_1, v_2) \blacktriangleright \overline{(v_1, v_2)} \stackrel{\triangle}{=} (v_1 \blacktriangleright \overline{v_1'}), (v_2 \blacktriangleright \overline{v_2'})$$

$$\langle \{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\}, \lambda x\ e \rangle \blacktriangleright \overline{\langle \{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\}, \lambda x\ e \rangle} \stackrel{\triangle}{=}$$

$$\langle \{x_1 \mapsto (v_1 \blacktriangleright \overline{v_1'}), \ldots, x_n \mapsto (v_n \blacktriangleright \overline{v_n'})\}, \lambda x\ e \rangle$$

$$\overrightarrow{u} \triangleq \lambda(v \blacktriangleright \overline{v})\ ((u\ v) \blacktriangleright ((\mathcal{D}\ u\ v) \times \overline{v}))$$

$$\overrightarrow{b} \triangleq \lambda((v_1, v_2) \blacktriangleright \overline{(v_1, v_2)})\ (b\ (v_1, v_2)) \blacktriangleright (((\mathcal{D}_1\ b\ (v_1, v_2)) \times \overline{v_1}) + ((\mathcal{D}_2\ b\ (v_1, v_2)) \times \overline{v_2}))$$

$$\overrightarrow{p} \triangleq \lambda(v \blacktriangleright \overline{v})\ \overrightarrow{\mathcal{J}}\ (p\ v)$$

$$\overrightarrow{q} \triangleq \lambda((v_1, v_2) \blacktriangleright \overline{(v_1, v_2)})\ \overrightarrow{\mathcal{J}}\ (q\ (v_1, v_2))$$

```
(define (bundle v v-tangent)
 (cond ((or (and (null? v) (null? v-tangent))
            (and (boolean? v) (eq? v v-tangent))
            (and (real? v) (real? v-tangent)))
        (make-bundle v v-tangent))
       ((and (equal? v +) (equal? v-tangent +))
        (lambda (v-forward)
         (bundle (+ (car (primal v-forward))
                    (cdr (primal v-forward)))
                 (+ (car (tangent v-forward))
                    (cdr (tangent v-forward))))))
       ;; ...
       ((and (bundle? v) (bundle? v-tangent))
        (make-bundle
         (bundle (bundle-primal v) (bundle-primal v-tangent))
         (bundle (bundle-tangent v) (bundle-tangent v-tangent))))
       ((and (pair? v) (pair? v-tangent))
        (cons (bundle (car v) (car v-tangent))
              (bundle (cdr v) (cdr v-tangent))))
       ((and (procedure? v) (procedure? v-tangent))
        (map-closure bundle v v-tangent))
       (else (fuck-up))))
```

$\textsc{Primal}\ \overrightarrow{v} \stackrel{\triangle}{=} v \qquad$ when v is a primitive

$\textsc{Primal}\ (v \rhd \overrightarrow{v'}) \stackrel{\triangle}{=} v$

$\textsc{Primal}\ ((v_1 \blacktriangleright \overrightarrow{v_1'}), (v_2 \blacktriangleright \overrightarrow{v_2'})) \stackrel{\triangle}{=} v_1, v_2$

$\textsc{Primal}\ \langle \{x_1 \mapsto (v_1 \blacktriangleright \overrightarrow{v_1'}), \ldots, x_n \mapsto (v_n \blacktriangleright \overrightarrow{v_n'})\}, \lambda x\ e \rangle \stackrel{\triangle}{=}$
$\langle \{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\}, \lambda x\ e \rangle$

```
(define (primal v)
 (cond
  ((equal? v (j* +)) +)
  ;; ...
  ((bundle? v) (bundle-primal v))
  ((pair? v)
   (cons (primal (car v)) (primal (cdr v))))
  ((procedure? v) (map-closure primal v))
  (else (fuck-up))))
```

$\textsc{Tangent}\ \overrightarrow{v} \stackrel{\triangle}{=} v \qquad$ when v is a primitive

$\textsc{Tangent}\ (v \rhd \overrightarrow{v'}) \stackrel{\triangle}{=} \overrightarrow{v'}$

$\textsc{Tangent}\ ((v_1 \blacktriangleright \overrightarrow{v_1'}), (v_2 \blacktriangleright \overrightarrow{v_2'})) \stackrel{\triangle}{=} \overrightarrow{v_1'}, \overrightarrow{v_2'}$

$\textsc{Tangent}\ \langle \{x_1 \mapsto (v_1 \blacktriangleright \overrightarrow{v_1'}), \ldots, x_n \mapsto (v_n \blacktriangleright \overrightarrow{v_n'})\}, \lambda x\ e \rangle \stackrel{\triangle}{=}$
$\langle \{x_1 \mapsto \overrightarrow{v_1'}, \ldots, x_n \mapsto \overrightarrow{v_n'}\}, \lambda x\ e \rangle$

```
(define (tangent v)
 (cond
  ((equal? v (j* +)) +)
  ;; ...
  ((bundle? v) (bundle-tangent v))
  ((pair? v)
   (cons (tangent (car v)) (tangent (cdr v))))
  ((procedure? v) (map-closure tangent v))
  (else (fuck-up))))
```

$$\mathbf{0}\ v \stackrel{\triangle}{=} 0 \qquad \text{when } v \in \mathbb{R}$$

$$\mathbf{0}\ v \stackrel{\triangle}{=} v \qquad \text{when } v \text{ is a discrete scalar}$$

$$\mathbf{0}\ (v \rhd \overline{v'}) \stackrel{\triangle}{=} (\mathbf{0}\ v) \rhd (\mathbf{0}\ \overline{v'})$$

$$\mathbf{0}\ (v_1, v_2) \stackrel{\triangle}{=} (\mathbf{0}\ v_1), (\mathbf{0}\ v_2)$$

$$\mathbf{0}\ \langle \{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\}, \lambda x\ e \rangle \stackrel{\triangle}{=} \langle \{x_1 \mapsto (\mathbf{0}\ v_1), \ldots, x_n \mapsto (\mathbf{0}\ v_n)\}, \lambda x\ e \rangle$$

```
(define (zero v)
 (cond
  ((real? v) 0)
  ((null? v) v)
  ((boolean? v) v)
  ((bundle? v)
   (make-bundle (zero (bundle-primal v))
                (zero (bundle-tangent v))))
  ((pair? v) (cons (zero (car v)) (zero (cdr v))))
  ((procedure? v) (map-closure zero v))
  (else (fuck-up))))
```

$$\overrightarrow{\mathcal{J}}\ v \stackrel{\triangle}{=} v \blacktriangleright (\mathbf{0}\ v)$$

```
(define (j* v) (bundle v (zero v)))
```

A Xillion Implementations of AD

A Xillion Implementations of AD

MAPLE: GRADIENT (Monagan & Neuenschwander, 1993)

A Xillion Implementations of AD

MAPLE: GRADIENT (Monagan & Neuenschwander, 1993)

FORTRAN: ADIFOR (Bischof *et al.*, 1996)

A Xillion Implementations of AD

MAPLE: GRADIENT (Monagan & Neuenschwander, 1993)

FORTRAN: ADIFOR (Bischof *et al.*, 1996)

C++: FADBAD++ (Bendtsen & Stauning, 1996)

A Xillion Implementations of AD

MAPLE: GRADIENT (Monagan & Neuenschwander, 1993)

FORTRAN: ADIFOR (Bischof *et al.*, 1996)

C++: FADBAD++ (Bendtsen & Stauning, 1996)

C: ADIC (Bischof *et al.*, 1997)

A Xillion Implementations of AD

MAPLE: GRADIENT (Monagan & Neuenschwander, 1993)

FORTRAN: ADIFOR (Bischof *et al.*, 1996)

C++: FADBAD++ (Bendtsen & Stauning, 1996)

C: ADIC (Bischof *et al.*, 1997)

HASKELL: (Karczmarczuk, 1998, 1999, 2001; Nilsson, 2003)

A Xillion Implementations of AD

MAPLE: GRADIENT (Monagan & Neuenschwander, 1993)

FORTRAN: ADIFOR (Bischof *et al.*, 1996)

C++: FADBAD++ (Bendtsen & Stauning, 1996)

C: ADIC (Bischof *et al.*, 1997)

HASKELL: (Karczmarczuk, 1998, 1999, 2001; Nilsson, 2003)

SCHEME: SCMUTILS (Sussman *et al.*, 2001)

A Xillion Implementations of AD

MAPLE:	GRADIENT (Monagan & Neuenschwander, 1993)
FORTRAN:	ADIFOR (Bischof *et al.*, 1996)
C++:	FADBAD++ (Bendtsen & Stauning, 1996)
C:	ADIC (Bischof *et al.*, 1997)
HASKELL:	(Karczmarczuk, 1998, 1999, 2001; Nilsson, 2003)
SCHEME:	SCMUTILS (Sussman *et al.*, 2001)
MATLAB:	ADIMAT (Bischof *et al.*, 2003)

A Xillion Implementations of AD

MAPLE:	GRADIENT (Monagan & Neuenschwander, 1993)
FORTRAN:	ADIFOR (Bischof *et al.*, 1996)
C++:	FADBAD++ (Bendtsen & Stauning, 1996)
C:	ADIC (Bischof *et al.*, 1997)
HASKELL:	(Karczmarczuk, 1998, 1999, 2001; Nilsson, 2003)
SCHEME:	SCMUTILS (Sussman *et al.*, 2001)
MATLAB:	ADIMAT (Bischof *et al.*, 2003)
	⋮

A Xillion Implementations of AD

MAPLE: GRADIENT (Monagan & Neuenschwander, 1993)

FORTRAN: ADIFOR (Bischof *et al.*, 1996)

C++: FADBAD++ (Bendtsen & Stauning, 1996)

C: ADIC (Bischof *et al.*, 1997)

HASKELL: (Karczmarczuk, 1998, 1999, 2001; Nilsson, 2003)

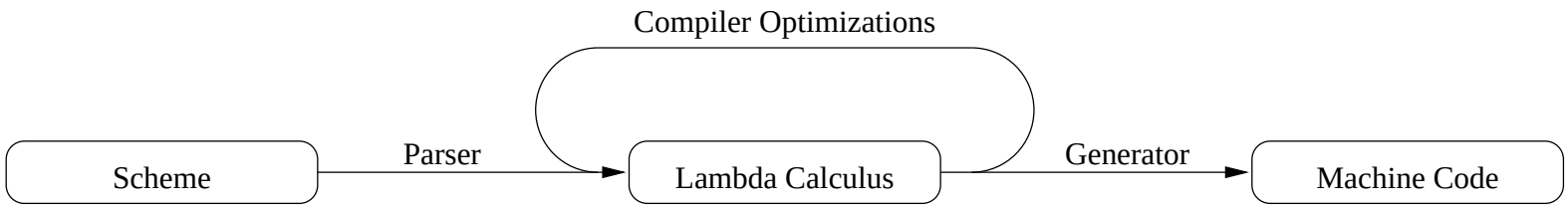
SCHEME: SCMUTILS (Sussman *et al.*, 2001)

MATLAB: ADIMAT (Bischof *et al.*, 2003)

⋮

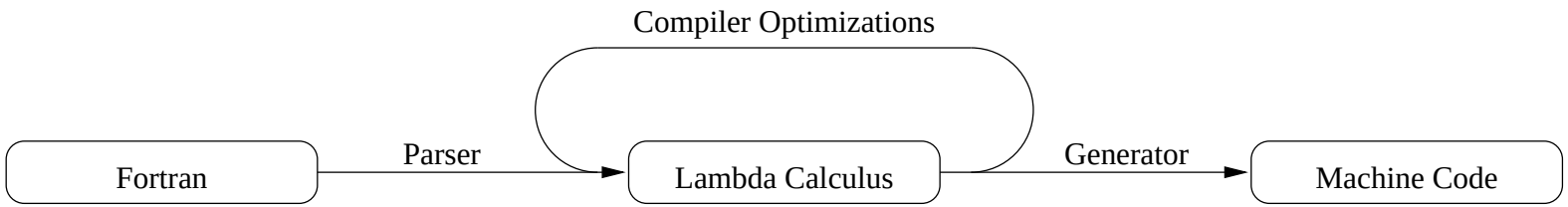
`http://www.autodiff.org`

Lambda the Ultimate Intermediate Language



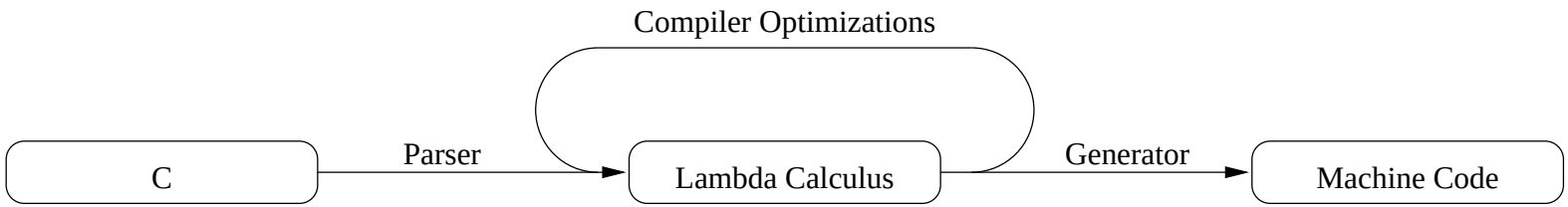
Steele, Jr., G. L. and Sussman, G. J. (1976). *Lambda, the Ultimate Imperative*, MIT AI memo 353.

Lambda the Ultimate Intermediate Language

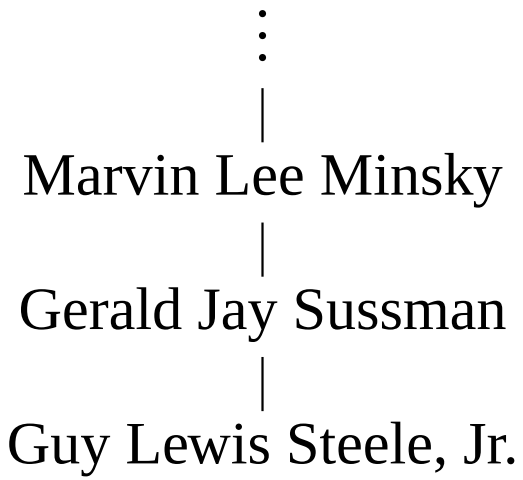


Steele, Jr., G. L. and Sussman, G. J. (1976). *Lambda, the Ultimate Imperative*, MIT AI memo 353.

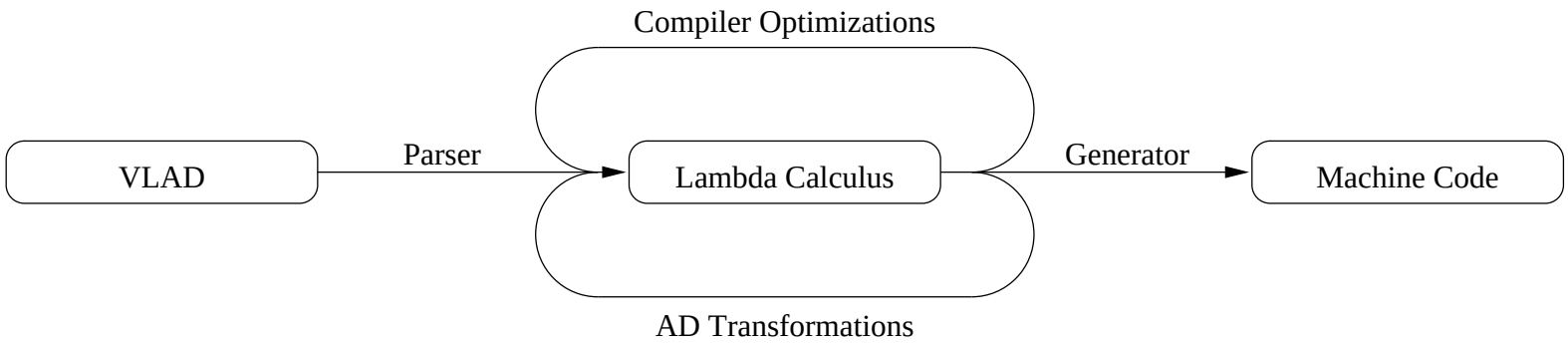
Lambda the Ultimate Intermediate Language



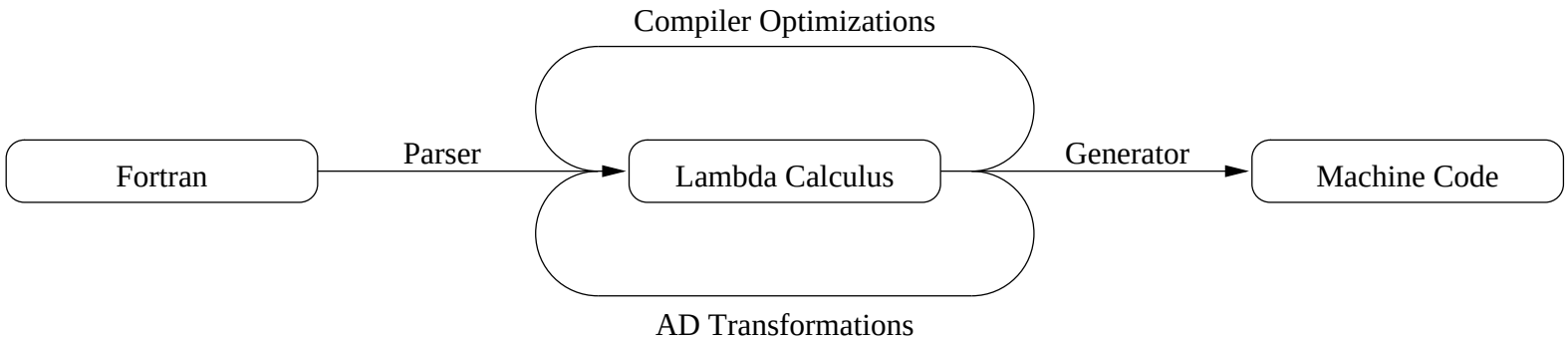
Steele, Jr., G. L. and Sussman, G. J. (1976). *Lambda, the Ultimate Imperative*, MIT AI memo 353.


Marvin Lee Minsky
Gerald Jay Sussman
Guy Lewis Steele, Jr.

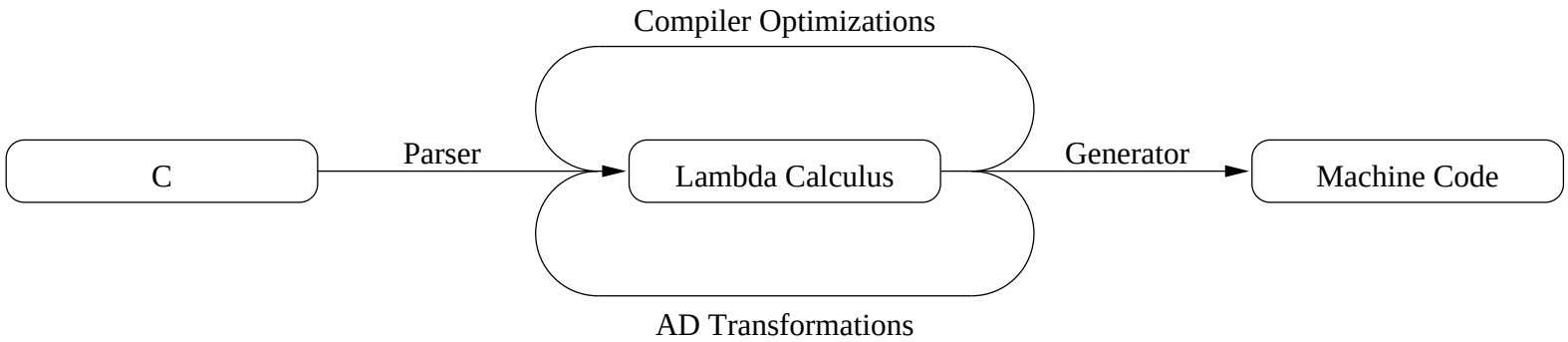
Lambda the Ultimate Intermediate Language *for AD*



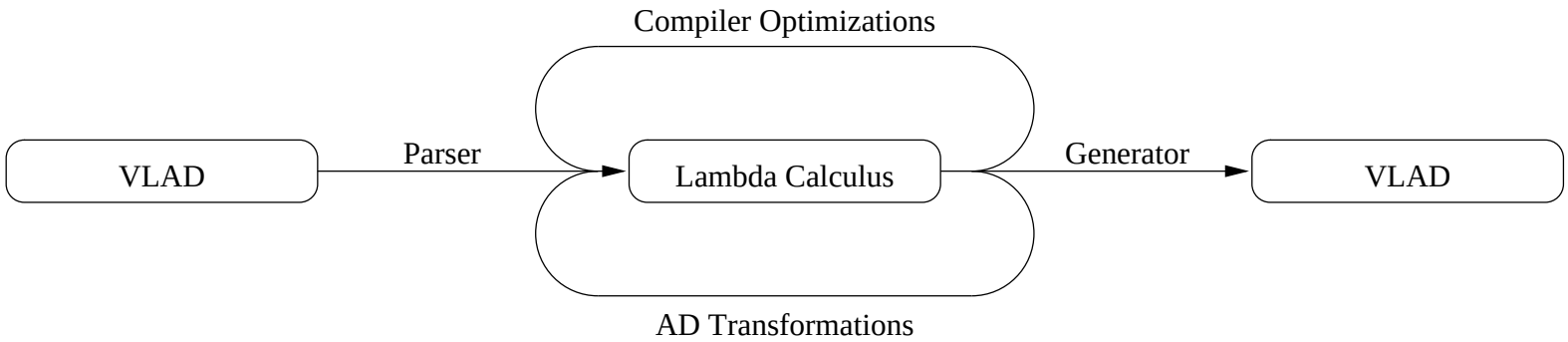
Lambda the Ultimate Intermediate Language *for AD*



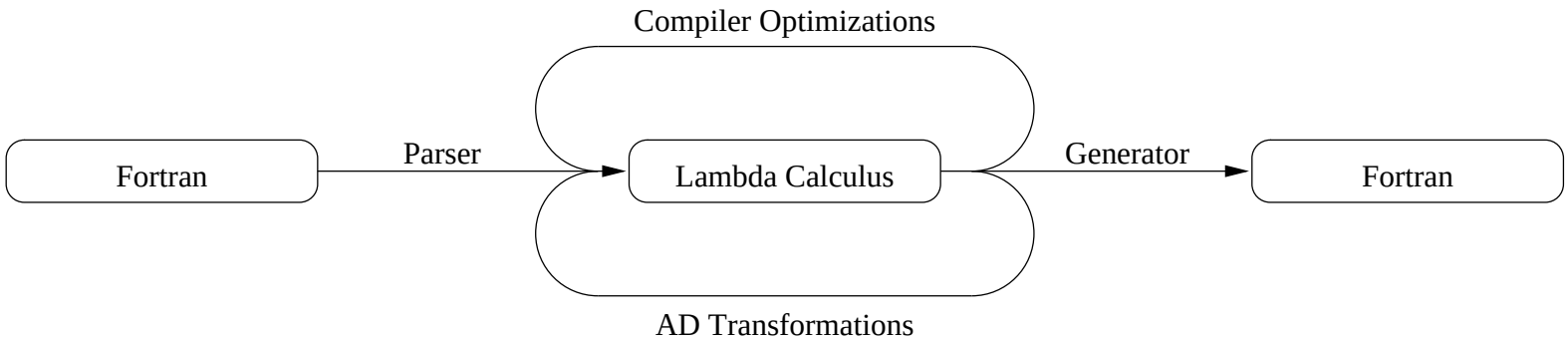
Lambda the Ultimate Intermediate Language *for AD*



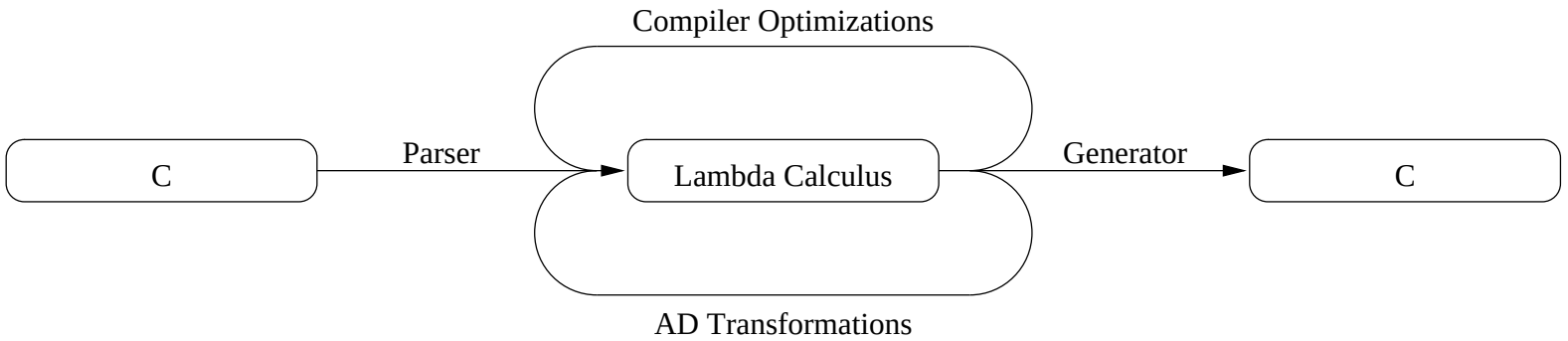
Lambda the Ultimate Intermediate Language *for AD*



Lambda the Ultimate Intermediate Language *for AD*

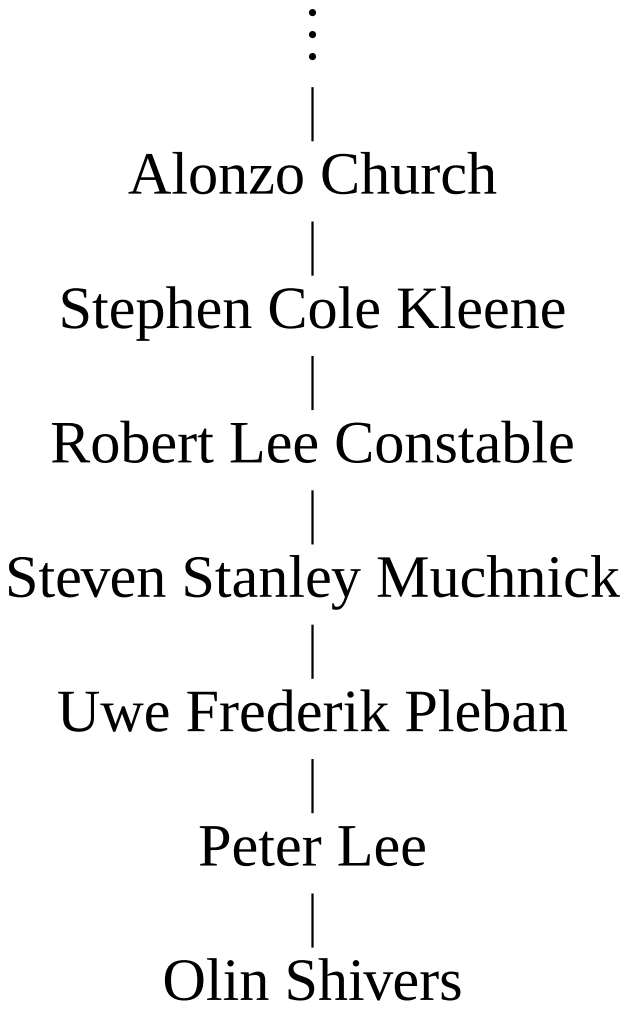


Lambda the Ultimate Intermediate Language *for AD*



Polyvariant flow analysis can (in most cases) eliminate run-time invocation of `map-closure` in `bundle`, `primal`, `tangent`, and `zero`.

Shivers, III, O. G. (1991). *Control-Flow Analysis of Higher-Order Languages* ***or*** *Taming Lambda*, Ph.D. thesis, CMU.


Alonzo Church
Stephen Cole Kleene
Robert Lee Constable
Steven Stanley Muchnick
Uwe Frederik Pleban
Peter Lee
Olin Shivers

Derivatives

$$\mathcal{D}\ f\ x \stackrel{\triangle}{=} \textsc{Tangent}\ ((\overrightarrow{\mathcal{J}}\ f)\ (x \blacktriangleright 1))$$

Roots using Newton-Raphson

$$\text{ROOT}\ f\ x_0\ \epsilon \stackrel{\triangle}{=} \textbf{let}\ x' \stackrel{\triangle}{=} x_0 - \frac{f\ x_0}{\mathcal{D}\ f\ x_0}$$
$$\textbf{in if}\ |x_0 - x'| \leq \epsilon\ \textbf{then}\ x_0\ \textbf{else}\ \text{ROOT}\ f\ x'\ \epsilon$$

Univariate Minimizer

Line Search

$$\textsc{LineSearch}\ f\ x_0\ \epsilon \stackrel{\triangle}{=} \textsc{Root}\ (\mathcal{D}\ f)\ x_0\ \epsilon$$

Gradients

$$\nabla f\ x \stackrel{\triangle}{=} \textbf{let}\ n \stackrel{\triangle}{=} \textsc{Length}\ x$$
$$\textbf{in}\ \textsc{Map}\ (\lambda i\ \textsc{Tangent}\ ((\overrightarrow{\mathcal{J}}\ f)\ (x \blacktriangleright e_{i,n})))\ (\iota\ n)$$

Multivariate Minimizer

Gradient Descent

$$\begin{array}{l}
\textsc{GradientDescent}\ f\ x_0\ \epsilon \stackrel{\triangle}{=} \\
\quad \textbf{let}\ g \stackrel{\triangle}{=} \nabla f\ x_0 \\
\quad \textbf{in if}\ \|g\| \leq \epsilon \\
\qquad \textbf{then}\ x_0 \\
\qquad \textbf{else}\ \textsc{GradientDescent} \\
\qquad\quad f\ (x_0 + ((\textsc{LineSearch}\ (\lambda k\ f\ (x_0 + (k \times g)))\ 0\ \epsilon) \times g))\ \epsilon
\end{array}$$

Saddle Points

Continuous Two-Person Zero Sum Games

$$\mathbf{x} : \mathbb{R}^m$$
$$\mathbf{y} : \mathbb{R}^n$$
$$\textsc{Payoff} : \mathbb{R}^m \times \mathbb{R}^n \rightarrow \mathbb{R}$$
$$\min_{\mathbf{x}} \max_{\mathbf{y}} \textsc{Payoff}\ (\mathbf{x}, \mathbf{y})$$

$$(\mathbf{x}^*, \mathbf{y}^*) = \textbf{let}\ \mathbf{x}^* \stackrel{\triangle}{=} \textsc{GradientDescent}\ (\lambda \mathbf{x}\ \textsc{GradientDescent}\ (\lambda \mathbf{y}\ (-\textsc{Payoff}\ (\mathbf{x}, \mathbf{y})))\ \mathbf{y}_0\ \epsilon)\ \mathbf{x}_0\ \epsilon$$
$$\textbf{in}\ (\mathbf{x}^*, (\textsc{GradientDescent}\ (\lambda \mathbf{y}\ (-\textsc{Payoff}\ (\mathbf{x}^*, \mathbf{y})))\ \mathbf{y}_0\ \epsilon))$$

von Neumann, J. and Morgenstern, O. (1944). *Theory of Games and Economic Behavior*. Princeton University Press, Princeton, NJ.

Carl Gauss

|

Christoph Gudermann

|

Karl Weierstrass

|

Hermann Schwarz

|

Leopold Fejér

|

John von Neumann

Neural Nets

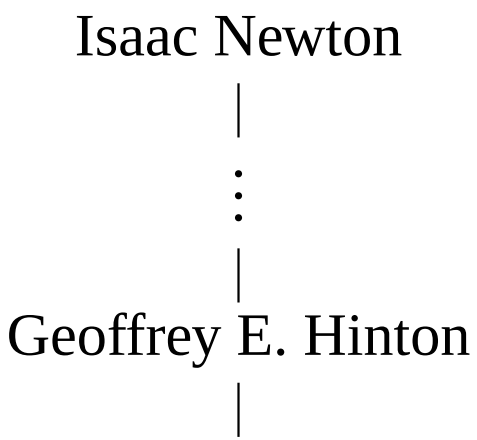
$$\textsc{Neuron}\ \mathbf{w}\ \mathbf{x} \stackrel{\triangle}{=} \textsc{Sigmoid}\ (\mathbf{w} \cdot \mathbf{x})$$

$$\textsc{NeuralNet}\ [\mathbf{w}, \mathbf{w}_1, \ldots, \mathbf{w}_m]\ \mathbf{x} \stackrel{\triangle}{=}$$
$$\quad \textsc{Neuron}\ \mathbf{w}\ [(\textsc{Neuron}\ \mathbf{w}_1\ \mathbf{x}), \ldots, (\textsc{Neuron}\ \mathbf{w}_m\ \mathbf{x})]$$

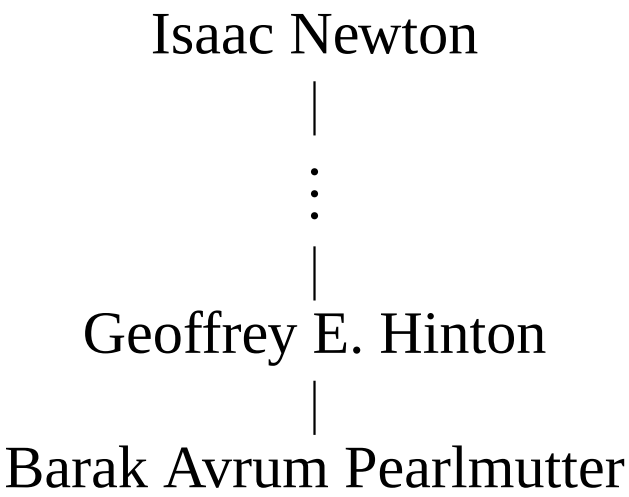
$$\textsc{Error}\ \mathbf{w} \stackrel{\triangle}{=}$$
$$\quad \|[y_1, \ldots, y_n] - [(\textsc{NeuralNet}\ \mathbf{w}\ \mathbf{x}_1), \ldots, (\textsc{NeuralNet}\ \mathbf{w}\ \mathbf{x}_n)]\|$$

$$\textsc{GradientDescent}\ \textsc{Error}\ \mathbf{w}_0\ \epsilon$$

Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986). Learning representations by back-propagating errors, *Nature*, **323**:533–6.



Newton, I. (1704). De quadratura curvarum, In *Optiks*, 1704 edition, appendix.



Newton, I. (1704). De quadratura curvarum, In *Optiks*, 1704 edition, appendix.

Work in Progress

Lambda: the ultimate calculus

Part of larger project to combine λ-calculus and AD.

Work in Progress

Lambda: the ultimate calculus

Part of larger project to combine λ-calculus and AD.

- include both forward-mode and reverse-mode operators

Work in Progress

Lambda: the ultimate calculus

Part of larger project to combine λ-calculus and AD.

- include both forward-mode and reverse-mode operators
- get numerically correct answers
- via correct transformation (time/space complexity)

Work in Progress

Lambda: the ultimate calculus

Part of larger project to combine λ-calculus and AD.

- include both forward-mode and reverse-mode operators
- get numerically correct answers
- via correct transformation (time/space complexity)
- make $\overrightarrow{\mathcal{J}}$ and $\overleftarrow{\mathcal{J}}$ first class
- make them apply to *all* (differentiable) programs
- handle nested application (self-application of $\overrightarrow{\mathcal{J}}$ and $\overleftarrow{\mathcal{J}}$)

Work in Progress

Lambda: the ultimate calculus

Part of larger project to combine λ-calculus and AD.

- include both forward-mode and reverse-mode operators
- get numerically correct answers
- via correct transformation (time/space complexity)
- make $\overrightarrow{\mathcal{J}}$ and $\overleftarrow{\mathcal{J}}$ first class
- make them apply to *all* (differentiable) programs
- handle nested application (self-application of $\overrightarrow{\mathcal{J}}$ and $\overleftarrow{\mathcal{J}}$)
- good puns

Work in Progress

Lambda: the ultimate calculus

Part of larger project to combine λ-calculus and AD.

- include both forward-mode and reverse-mode operators
- get numerically correct answers
- via correct transformation (time/space complexity)
- make $\overrightarrow{\mathcal{J}}$ and $\overleftarrow{\mathcal{J}}$ first class
- make them apply to *all* (differentiable) programs
- handle nested application (self-application of $\overrightarrow{\mathcal{J}}$ and $\overleftarrow{\mathcal{J}}$)
- good puns
  - programming language: Voice for FP + AD
    Functional Language for AD

Work in Progress

Lambda: the ultimate calculus

Part of larger project to combine λ-calculus and AD.

- include both forward-mode and reverse-mode operators
- get numerically correct answers
- via correct transformation (time/space complexity)
- make $\overrightarrow{\mathcal{J}}$ and $\overleftarrow{\mathcal{J}}$ first class
- make them apply to *all* (differentiable) programs
- handle nested application (self-application of $\overrightarrow{\mathcal{J}}$ and $\overleftarrow{\mathcal{J}}$)
- good puns
  - programming language: VLAD
    Functional Language for AD

Work in Progress

Lambda: the ultimate calculus

Part of larger project to combine λ-calculus and AD.

- include both forward-mode and reverse-mode operators
- get numerically correct answers
- via correct transformation (time/space complexity)
- make $\overrightarrow{\mathcal{J}}$ and $\overleftarrow{\mathcal{J}}$ first class
- make them apply to *all* (differentiable) programs
- handle nested application (self-application of $\overrightarrow{\mathcal{J}}$ and $\overleftarrow{\mathcal{J}}$)
- good puns
  - programming language: VLAD
    Functional Language for AD
  - implementation: high-performance SCHEME + AD

Work in Progress

Lambda: the ultimate calculus

Part of larger project to combine λ-calculus and AD.

- include both forward-mode and reverse-mode operators
- get numerically correct answers
- via correct transformation (time/space complexity)
- make $\overrightarrow{\mathcal{J}}$ and $\overleftarrow{\mathcal{J}}$ first class
- make them apply to *all* (differentiable) programs
- handle nested application (self-application of $\overrightarrow{\mathcal{J}}$ and $\overleftarrow{\mathcal{J}}$)
- good puns
  - programming language: VLAD
    Functional Language for AD
  - implementation: STALIN + AD

Work in Progress

Lambda: the ultimate calculus

Part of larger project to combine λ-calculus and AD.

- include both forward-mode and reverse-mode operators
- get numerically correct answers
- via correct transformation (time/space complexity)
- make $\overrightarrow{\mathcal{J}}$ and $\overleftarrow{\mathcal{J}}$ first class
- make them apply to *all* (differentiable) programs
- handle nested application (self-application of $\overrightarrow{\mathcal{J}}$ and $\overleftarrow{\mathcal{J}}$)
- good puns
  - programming language: VLAD
    Functional Language for AD
  - implementation: STALIN + gradients

Work in Progress

Lambda: the ultimate calculus

Part of larger project to combine λ-calculus and AD.

- include both forward-mode and reverse-mode operators
- get numerically correct answers
- via correct transformation (time/space complexity)
- make $\overrightarrow{\mathcal{J}}$ and $\overleftarrow{\mathcal{J}}$ first class
- make them apply to *all* (differentiable) programs
- handle nested application (self-application of $\overrightarrow{\mathcal{J}}$ and $\overleftarrow{\mathcal{J}}$)
- good puns
  - programming language: VLAD
    Functional Language for AD
  - implementation: STALIN $+ \nabla$

Work in Progress

Lambda: the ultimate calculus

Part of larger project to combine λ-calculus and AD.

- include both forward-mode and reverse-mode operators
- get numerically correct answers
- via correct transformation (time/space complexity)
- make $\overrightarrow{\mathcal{J}}$ and $\overleftarrow{\mathcal{J}}$ first class
- make them apply to *all* (differentiable) programs
- handle nested application (self-application of $\overrightarrow{\mathcal{J}}$ and $\overleftarrow{\mathcal{J}}$)
- good puns
  - programming language: VLAD
    Functional Language for AD
  - implementation: STALIN∇

Work in Progress

Lambda: the ultimate calculus

Part of larger project to combine λ-calculus and AD.

- include both forward-mode and reverse-mode operators
- get numerically correct answers
- via correct transformation (time/space complexity)
- make $\overrightarrow{\mathcal{J}}$ and $\overleftarrow{\mathcal{J}}$ first class
- make them apply to *all* (differentiable) programs
- handle nested application (self-application of $\overrightarrow{\mathcal{J}}$ and $\overleftarrow{\mathcal{J}}$)
- good puns
  - programming language: VLAD
    Functional Language for AD
  - implementation: STALIN∇
- manuscripts and code:
  `http://www.bcl.hamilton.ie/~qobi/stalingrad/`