

Nesting, Variable Capture, Programming Language Theory, and AD

Jeffrey Mark Siskind
qobi@purdue.edu

School of Electrical and Computer Engineering
Purdue University

2nd European Workshop on Automatic Differentiation
18 November 2005

Joint work with Barak A. Pearlmutter.

Outline

- 1 Lambda Calculus
- 2 Differential Calculus in Lambda-Calculus Notation
- 3 Essence of the Derivation of Functional Reverse Mode
- 4 AD in Lambda-Calculus Notation
- 5 An Example
- 6 Benefits of this Approach

Outline

- 1 Lambda Calculus
- 2 Differential Calculus in Lambda-Calculus Notation
- 3 Essence of the Derivation of Functional Reverse Mode
- 4 AD in Lambda-Calculus Notation
- 5 An Example
- 6 Benefits of this Approach

Higher-Order Functions

$$\text{FOLD } (m, n, u, b, i) \triangleq \begin{array}{l} \mathbf{if } m > n \\ \mathbf{then } i \\ \mathbf{else } b ((u \ m), (\text{FOLD } (m + 1, n, u, b, i))) \end{array}$$

Higher-Order Functions

$\text{FOLD } (m, n, u, b, i) \triangleq$ **if** $m > n$
 then i
 else $b ((u\ m), (\text{FOLD } (m + 1, n, u, b, i)))$

$$\sum_{i=m}^n \sin i$$

Higher-Order Functions

$$\text{FOLD } (m, n, u, b, i) \triangleq \begin{array}{l} \mathbf{if } m > n \\ \mathbf{then } i \\ \mathbf{else } b ((u \ m), (\text{FOLD } (m + 1, n, u, b, i))) \end{array}$$

$$\sum_{i=m}^n \sin i : \text{FOLD } (m, n, \sin, +, 0)$$

Higher-Order Functions

$$\text{FOLD } (m, n, u, b, i) \triangleq \begin{array}{l} \text{if } m > n \\ \text{then } i \\ \text{else } b ((u \ m), (\text{FOLD } (m + 1, n, u, b, i))) \end{array}$$

$$\sum_{i=m}^n \cos i : \text{FOLD } (m, n, \cos, +, 0)$$

Higher-Order Functions

$$\text{FOLD } (m, n, u, b, i) \triangleq \begin{array}{l} \mathbf{if } m > n \\ \quad \mathbf{then } i \\ \quad \mathbf{else } b ((u \ m), (\text{FOLD } (m + 1, n, u, b, i))) \end{array}$$

$$\prod_{i=m}^n \text{sin } i : \text{FOLD } (m, n, \text{sin}, \times, 1)$$

Lambda Expressions

Anonymous Functions

$$\sum_{i=m}^n i^2$$

Lambda Expressions

Anonymous Functions

$$\sum_{i=m}^n i^2$$
$$\text{SQR } i \triangleq i \times i$$

Lambda Expressions

Anonymous Functions

$$\sum_{i=m}^n i^2 = \text{FOLD } (m, n, \text{SQR}, +, 0)$$

$$\text{SQR } i \triangleq i \times i$$

Lambda Expressions

Anonymous Functions

$$\sum_{i=m}^n i^2 = \text{FOLD } (m, n, (\lambda i \ i \times i), +, 0)$$

Nesting, Free Variables, and Closures

$$(\lambda x 2 \times x) 3 = 6$$

Nesting, Free Variables, and Closures

$$(\lambda x 2 \times x) 3 = 6$$

$$((\lambda x \lambda y x + y) 3) 4 = 7$$

Nesting, Free Variables, and Closures

$$(\lambda x 2 \times x) 3 = 6$$

$$(\lambda x \lambda y x + y) 3 = ?$$

Nesting, Free Variables, and Closures

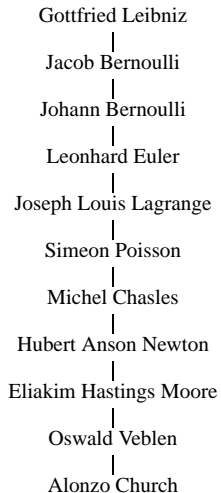
$$(\lambda x 2 \times x) 3 = 6$$

$$(\lambda x \lambda y x + y) 3 = \langle \{x \mapsto 3\}, \lambda y x + y \rangle$$

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.

(p. 1 ¶4)

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



Outline

- 1 Lambda Calculus
- 2 Differential Calculus in Lambda-Calculus Notation**
- 3 Essence of the Derivation of Functional Reverse Mode
- 4 AD in Lambda-Calculus Notation
- 5 An Example
- 6 Benefits of this Approach

Derivatives

$$\frac{dax^2}{dx} \rightsquigarrow 2ax$$

Derivatives

$$\frac{dax^2}{dx} \rightsquigarrow 2ax$$

$$\frac{d}{dx} : \underbrace{f}_{\mathbb{R} \rightarrow \mathbb{R}} \mapsto \underbrace{f'}_{\mathbb{R} \rightarrow \mathbb{R}}$$

Derivatives

$$\frac{dax^2}{dx} \rightsquigarrow 2ax$$

$$\frac{d}{dx} : \underbrace{f}_{\mathbb{R} \rightarrow \mathbb{R}} \mapsto \underbrace{f'}_{\mathbb{R} \rightarrow \mathbb{R}}$$

$$\frac{d}{dx} : (\mathbb{R} \rightarrow \mathbb{R}) \rightarrow (\mathbb{R} \rightarrow \mathbb{R})$$

Derivatives

$$\frac{dax^2}{dx} \rightsquigarrow 2ax$$

$$\frac{d}{dx} : \underbrace{f}_{\mathbb{R} \rightarrow \mathbb{R}} \mapsto \underbrace{f'}_{\mathbb{R} \rightarrow \mathbb{R}}$$

$$\frac{d}{dx} : (\mathbb{R} \rightarrow \mathbb{R}) \rightarrow (\mathbb{R} \rightarrow \mathbb{R})$$

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

Derivatives

$$\frac{dax^2}{dx} \rightsquigarrow 2ax$$

$$\frac{d}{dx} : \underbrace{f}_{\mathbb{R} \rightarrow \mathbb{R}} \mapsto \underbrace{f'}_{\mathbb{R} \rightarrow \mathbb{R}}$$

$$\frac{d}{dx} : (\mathbb{R} \rightarrow \mathbb{R}) \rightarrow (\mathbb{R} \rightarrow \mathbb{R})$$

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

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

Partial Derivatives

$$\frac{\partial ax^2y^3}{\partial x}$$

$$\frac{\partial ax^2y^3}{\partial y}$$

Partial Derivatives

$$\frac{\partial ax^2y^3}{\partial x}$$

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

$$\frac{\partial ax^2y^3}{\partial y}$$

$$\mathcal{D} \lambda y ax^2y^3$$

Partial Derivatives

$$\frac{\partial ax^2y^3}{\partial x}$$

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

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

$$\frac{\partial ax^2y^3}{\partial y}$$

$$\mathcal{D} \lambda y ax^2y^3$$

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

Partial Derivatives

$$\frac{\partial ax^2y^3}{\partial x}$$

$$\frac{\partial ax^2y^3}{\partial y}$$

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

$$\mathcal{D} \lambda y ax^2y^3$$

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

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

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

Partial Derivatives

$$\frac{\partial ax^2y^3}{\partial x}$$

$$\frac{\partial ax^2y^3}{\partial y}$$

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

$$\mathcal{D} \lambda y ax^2y^3$$

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

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

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

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

Partial Derivatives

$$\frac{\partial ax^2y^3}{\partial x}$$

$$\frac{\partial ax^2y^3}{\partial y}$$

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

$$\mathcal{D} \lambda y ax^2y^3$$

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

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

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

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

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

Gradients

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

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

Jacobians

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

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

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

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

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, . . .*

Outline

- 1 Lambda Calculus
- 2 Differential Calculus in Lambda-Calculus Notation
- 3 Essence of the Derivation of Functional Reverse Mode**
- 4 AD in Lambda-Calculus Notation
- 5 An Example
- 6 Benefits of this Approach

Reverse Mode on Imperative Programs

 \vdots $x_2 \quad := \quad u \ x_1$ $x_4 \quad := \quad b \ (x_2, x_3)$ \vdots

Reverse Mode on Imperative Programs

$$\begin{array}{rcl}
 & \vdots & \\
 x_2 & := & u x_1 \\
 & & \\
 x_4 & := & b(x_2, x_3) \\
 & \vdots & \\
 & \vdots & \\
 \overline{x_2} & +:= & (\mathcal{D}_1 b(x_2, x_3)) \times \overline{x_4} \\
 \overline{x_3} & +:= & (\mathcal{D}_2 b(x_2, x_3)) \times \overline{x_4} \\
 & & \\
 \overline{x_1} & +:= & (\mathcal{D} u x_1) \times \overline{x_2} \\
 & \vdots &
 \end{array}
 \left. \begin{array}{l} \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \end{array} \right\} \begin{array}{l} \text{forward phase} \\ \\ \\ \\ \\ \\ \\ \text{reverse phase} \end{array}$$

Reverse Mode on Imperative Programs

$$\begin{array}{rcl}
 & \vdots & \\
 x_2 & := & u x_1 \\
 & \vdots & \\
 x_4 & := & b(x_2, x_3) \\
 & \vdots & \\
 & \vdots & \\
 \overline{x_2} & +:= & (\mathcal{D}_1 b(x_2, x_3)) \times \overline{x_4} \\
 \overline{x_3} & +:= & (\mathcal{D}_2 b(x_2, x_3)) \times \overline{x_4} \\
 & \vdots & \\
 \overline{x_1} & +:= & (\mathcal{D} u x_1) \times \overline{x_2} \\
 & \vdots &
 \end{array}
 \left. \begin{array}{l} \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \end{array} \right\} \begin{array}{l} \text{forward phase} \\ \\ \\ \\ \\ \\ \\ \text{reverse phase} \end{array}$$

Reverse Mode on Imperative Programs

$$\begin{array}{rcl}
 & \vdots & \\
 x_2 & := & u x_1 \\
 & & \\
 \color{red}{x_2} & := & b(x_2, x_3) \\
 & \vdots & \\
 & \vdots & \\
 \overline{x_2} & +:= & (\mathcal{D}_1 b(x_2, x_3)) \times \overline{x_2} \\
 \overline{x_3} & +:= & (\mathcal{D}_2 b(x_2, x_3)) \times \overline{x_2} \\
 & & \\
 \overline{x_1} & +:= & (\mathcal{D} u x_1) \times \overline{x_2} \\
 & \vdots &
 \end{array}
 \left. \begin{array}{l} \\ \\ \\ \\ \\ \\ \\ \\ \\ \end{array} \right\} \begin{array}{l} \text{forward phase} \\ \\ \\ \\ \\ \\ \\ \text{reverse phase} \end{array}$$

Reverse Mode on Imperative Programs

$\begin{array}{l} \vdots \\ \text{PUSH } x_1 \\ x_2 := u x_1 \\ \text{PUSH } x_2 \\ \text{PUSH } x_3 \\ x_2 := b(x_2, x_3) \\ \vdots \\ \vdots \\ \text{POP } x_3 \\ \text{POP } x_2 \\ \overline{x_2} += (\mathcal{D}_1 b(x_2, x_3)) \times \overline{x_2} \\ \overline{x_3} += (\mathcal{D}_2 b(x_2, x_3)) \times \overline{x_2} \\ \text{POP } x_1 \\ \overline{x_1} += (\mathcal{D} u x_1) \times \overline{x_2} \\ \vdots \end{array}$	}	<i>forward phase</i>
$\begin{array}{l} \vdots \\ \vdots \\ \text{POP } x_3 \\ \text{POP } x_2 \\ \overline{x_2} += (\mathcal{D}_1 b(x_2, x_3)) \times \overline{x_2} \\ \overline{x_3} += (\mathcal{D}_2 b(x_2, x_3)) \times \overline{x_2} \\ \text{POP } x_1 \\ \overline{x_1} += (\mathcal{D} u x_1) \times \overline{x_2} \\ \vdots \end{array}$	}	<i>reverse phase</i>

Notation

In the following slides, I use \mathbf{x} , \mathbf{y} , \mathbf{x}_i , and \mathbf{y}_i to denote tuples of scalar variables, i.e. (x_{47}, x_{19}, x_{33}) .

Notation

In the following slides, I use \mathbf{x} , \mathbf{y} , \mathbf{x}_i , and \mathbf{y}_i to denote tuples of scalar variables, i.e. (x_{47}, x_{19}, x_{33}) .

I use $\overline{\mathbf{x}}$, $\overline{\mathbf{y}}$, $\overline{\mathbf{x}}_i$, and $\overline{\mathbf{y}}_i$ to denote tuples of corresponding sensitivities of scalar variables, i.e. $(\overline{x_{47}}, \overline{x_{19}}, \overline{x_{33}})$.

Subroutines and Tapes

Unary Primitives

$$u : x \mapsto y$$

Subroutines and Tapes

Unary Primitives

$$u : x \mapsto y$$

$$\overleftarrow{u} : x \mapsto y \quad \triangleq \quad \begin{cases} \text{PUSH } x \\ y := u x \end{cases}$$

$$\overline{u} : \overleftarrow{y} \mapsto \overleftarrow{x} \quad \triangleq \quad \begin{cases} \text{POP } x \\ \overleftarrow{x} += (\mathcal{D} u x) \times \overleftarrow{y} \end{cases}$$

Subroutines and Tapes

Binary Primitives

$$b : (x, y) \mapsto z$$

Subroutines and Tapes

Binary Primitives

$$b : (x, y) \mapsto z$$

$$\underline{b} : (x, y) \mapsto z \quad \triangleq \quad \begin{cases} \text{PUSH } x \\ \text{PUSH } y \\ z := b(x, y) \end{cases}$$

$$\bar{b} : \overline{z} \mapsto (\overline{x}, \overline{y}) \quad \triangleq \quad \begin{cases} \text{POP } x \\ \text{POP } y \\ \overline{x} += (\mathcal{D}_1 b(x, y)) \times \overline{z} \\ \overline{y} += (\mathcal{D}_2 b(x, y)) \times \overline{z} \end{cases}$$

Subroutines and Tapes

User-Defined Functions

$$f : \mathbf{x} \mapsto \mathbf{y} \quad \triangleq \quad \left\{ \begin{array}{l} \mathbf{y}_1 := f_1 \mathbf{x}_1 \\ \vdots \\ \mathbf{y}_n := f_n \mathbf{x}_n \end{array} \right.$$

Subroutines and Tapes

User-Defined Functions

$$f : \mathbf{x} \mapsto \mathbf{y} \quad \triangleq \quad \left\{ \begin{array}{l} \mathbf{y}_1 := f_1 \mathbf{x}_1 \\ \vdots \\ \mathbf{y}_n := f_n \mathbf{x}_n \end{array} \right.$$

$$\overleftarrow{f} : \mathbf{x} \mapsto \mathbf{y} \quad \triangleq \quad \left\{ \begin{array}{l} \mathbf{y}_1 := \overleftarrow{f}_1 \mathbf{x}_1 \\ \vdots \\ \mathbf{y}_n := \overleftarrow{f}_n \mathbf{x}_n \end{array} \right.$$

$$\overline{f} : \overline{\mathbf{y}} \mapsto \overline{\mathbf{x}} \quad \triangleq \quad \left\{ \begin{array}{l} \overline{\mathbf{x}}_n + := \overline{f}_n \overline{\mathbf{y}}_n \\ \vdots \\ \overline{\mathbf{x}}_1 + := \overline{f}_1 \overline{\mathbf{y}}_1 \end{array} \right.$$

Representing the Tape as Function Arguments and Results

Unary Primitives

$$u : x \mapsto y$$

$$\overleftarrow{u} : x \mapsto (y, x) \quad \triangleq \quad \{ y := u x$$

$$\overline{u} : (x, \overline{y}) \mapsto \overline{x} \quad \triangleq \quad \{ \overline{x} \text{ } +:= \text{ } (\mathcal{D} u x) \times \overline{y}$$

Representing the Tape as Function Arguments and Results

Binary Primitives

$$b : (x, y) \mapsto z$$

$$\overleftarrow{b} : (x, y) \mapsto (z, (x, y)) \quad \triangleq \quad \{ z := b(x, y)$$

$$\overline{b} : ((x, y), \overline{z}) \mapsto (\overline{x}, \overline{y}) \quad \triangleq \quad \begin{cases} \overline{x} & +:= (\mathcal{D}_1 b(x, y)) \times \overline{z} \\ \overline{y} & +:= (\mathcal{D}_2 b(x, y)) \times \overline{z} \end{cases}$$

Representing the Tape as Function Arguments and Results

User-Defined Functions

$$f : \mathbf{x} \mapsto \mathbf{y} \quad \triangleq \quad \left\{ \begin{array}{l} \mathbf{y}_1 := f_1 \mathbf{x}_1 \\ \vdots \\ \mathbf{y}_n := f_n \mathbf{x}_n \end{array} \right.$$

$$\overleftarrow{f} : \mathbf{x} \mapsto (\mathbf{y}, (\mathbf{t}_1, \dots, \mathbf{t}_n)) \quad \triangleq \quad \left\{ \begin{array}{l} \mathbf{y}_1, \mathbf{t}_1 := \overleftarrow{f}_1 \mathbf{x}_1 \\ \vdots \\ \mathbf{y}_n, \mathbf{t}_n := \overleftarrow{f}_n \mathbf{x}_n \end{array} \right.$$

$$\overline{f} : ((\mathbf{t}_1, \dots, \mathbf{t}_n), \overline{\mathbf{y}}) \mapsto \overline{\mathbf{x}} \quad \triangleq \quad \left\{ \begin{array}{l} \overline{\mathbf{x}}_n + := \overline{f}_n (\mathbf{t}_n, \overline{\mathbf{y}}_n) \\ \vdots \\ \overline{\mathbf{x}}_1 + := \overline{f}_1 (\mathbf{t}_1, \overline{\mathbf{y}}_1) \end{array} \right.$$

Representing the Tape as Closures

Unary Primitives

$$u : x \mapsto y$$

$$\overleftarrow{u} : x \mapsto (y, \overline{u}) \triangleq \begin{cases} y & := u x \\ \overline{u} : \overleftarrow{y} \mapsto \overleftarrow{x} & \triangleq \{ \overleftarrow{x} \} + := (\mathcal{D} u x) \times \overleftarrow{y} \end{cases}$$

Representing the Tape as Closures

Binary Primitives

$$b : (x, y) \mapsto z$$

$$\overleftarrow{b} : (x, y) \mapsto (z, \overline{b}) \triangleq \begin{cases} z & := b(x, y) \\ \overline{b} : \overline{z} \mapsto (\overline{x}, \overline{y}) & \triangleq \begin{cases} \overline{x} & +:= (\mathcal{D}_1 b(x, y)) \times \overline{z} \\ \overline{y} & +:= (\mathcal{D}_2 b(x, y)) \times \overline{z} \end{cases} \end{cases}$$

Representing the Tape as Closures

User-Defined Functions

$$f : \mathbf{x} \mapsto \mathbf{y} \quad \triangleq \quad \begin{cases} \mathbf{y}_1 & := f_1 \mathbf{x}_1 \\ \vdots & \\ \mathbf{y}_n & := f_n \mathbf{x}_n \end{cases}$$

$$\overleftarrow{f} : \mathbf{x} \mapsto (\mathbf{y}, \overline{f}) \quad \triangleq \quad \begin{cases} \mathbf{y}_1, \overline{f}_1 & := \overleftarrow{f}_1 \mathbf{x}_1 \\ \vdots & \\ \mathbf{y}_n, \overline{f}_n & := \overleftarrow{f}_n \mathbf{x}_n \\ \overline{f} : \overline{\mathbf{y}} \mapsto \overline{\mathbf{x}} & \triangleq \begin{cases} \overline{\mathbf{x}}_n & + := \overline{f}_n \overline{\mathbf{y}}_n \\ \vdots & \\ \overline{\mathbf{x}}_1 & + := \overline{f}_1 \overline{\mathbf{y}}_1 \end{cases} \end{cases}$$

Details for Handling Closures Omitted

Outline

- 1 Lambda Calculus
- 2 Differential Calculus in Lambda-Calculus Notation
- 3 Essence of the Derivation of Functional Reverse Mode
- 4 AD in Lambda-Calculus Notation**
- 5 An Example
- 6 Benefits of this Approach

Traditional Formulation of AD as Transformations

Forward Mode: $\mathbb{R}^n \rightarrow \mathbb{R}^m \rightsquigarrow (\mathbb{R}^n \times \mathbb{R}^n) \rightarrow (\mathbb{R}^m \times \mathbb{R}^m)$

Reverse Mode: $\mathbb{R}^n \rightarrow \mathbb{R}^m \rightsquigarrow (\mathbb{R}^n \rightarrow (\mathbb{R}^m \times \mathbb{R}^l)) \times ((\mathbb{R}^m \times \mathbb{R}^l) \rightarrow \mathbb{R}^n)$

New Formulation of AD as Higher-Order Functions

Perturbation Types

$$\overline{\mathbf{null}} = \mathbf{null}$$

$$\overline{\mathbb{R}} = \mathbb{R}$$

$$\overline{\tau_1 \times \tau_2} = \overline{\tau_1} \times \overline{\tau_2}$$

$$\overline{\tau_1 \xrightarrow{\tau'_1, \dots, \tau'_n} \tau_2} = \overline{\tau_1} \times \dots \times \overline{\tau_n}$$

New Formulation of AD as Higher-Order Functions

Forward Types

$$\overrightarrow{\text{null}} = \text{null} \times \overrightarrow{\text{null}}$$

$$\overrightarrow{\mathbb{R}} = \mathbb{R} \times \overrightarrow{\mathbb{R}}$$

$$\overrightarrow{\tau_1 \times \tau_2} = \overrightarrow{\tau_1} \times \overrightarrow{\tau_2}$$

$$\overrightarrow{\tau_1 \xrightarrow{\tau'_1, \dots, \tau'_n} \tau_2} = \overrightarrow{\tau_1} \xrightarrow{\overrightarrow{\tau'_1}, \dots, \overrightarrow{\tau'_n}} \overrightarrow{\tau_2}$$

New Formulation of AD as Higher-Order Functions

Sensitivity Types

$$\overline{\mathbf{null}} = \mathbf{null}$$

$$\overline{\mathbb{R}} = \mathbb{R}$$

$$\overline{\tau_1 \times \tau_2} = \overline{\tau_1} \times \overline{\tau_2}$$

$$\overline{\tau_1 \xrightarrow{\tau'_1, \dots, \tau'_n} \tau_2} = \overline{\tau'_1} \times \dots \times \overline{\tau'_n}$$

New Formulation of AD as Higher-Order Functions

Reverse Types

$$\overleftarrow{\text{null}} = \text{null}$$

$$\overleftarrow{\mathbb{R}} = \mathbb{R}$$

$$\overleftarrow{\tau_1 \times \tau_2} = \overleftarrow{\tau_1} \times \overleftarrow{\tau_2}$$

$$\overleftarrow{\tau_1 \xrightarrow{\tau'_1, \dots, \tau'_n} \tau_2} = \overleftarrow{\tau_1} \xrightarrow{\overleftarrow{\tau'_1}, \dots, \overleftarrow{\tau'_n}} (\overleftarrow{\tau_2} \times (\overleftarrow{\tau_2} \rightarrow (\overleftarrow{\tau'_1} \times \dots \times \overleftarrow{\tau'_n}) \times \overleftarrow{\tau_1}))$$

New Formulation of AD as Higher-Order Functions

Forward Mode: $\vec{\mathcal{J}} : \tau \rightarrow \vec{\tau}$

Reverse Mode: $\overleftarrow{\mathcal{J}} : \tau \rightarrow \overleftarrow{\tau}$

Outline

- 1 Lambda Calculus
- 2 Differential Calculus in Lambda-Calculus Notation
- 3 Essence of the Derivation of Functional Reverse Mode
- 4 AD in Lambda-Calculus Notation
- 5 An Example**
- 6 Benefits of this Approach

Saddle Points

Continuous Two-Person Zero Sum Games

$$\mathbf{x} : \mathbb{R}^m$$

$$\mathbf{y} : \mathbb{R}^n$$

$$\text{PAYOFF} : \mathbb{R}^m \times \mathbb{R}^n \rightarrow \mathbb{R}$$

$$\min_{\mathbf{x}} \max_{\mathbf{y}} \text{PAYOFF}(\mathbf{x}, \mathbf{y})$$

Saddle Points

Continuous Two-Person Zero Sum Games

$$\mathbf{x} : \mathbb{R}^m$$

$$\mathbf{y} : \mathbb{R}^n$$

$$\text{PAYOFF} : \mathbb{R}^m \times \mathbb{R}^n \rightarrow \mathbb{R}$$

$$\min_{\mathbf{x}} \max_{\mathbf{y}} \text{PAYOFF}(\mathbf{x}, \mathbf{y})$$

$$(\mathbf{x}^*, \mathbf{y}^*) = \mathbf{let} \mathbf{x}^* \triangleq \text{ARGMIN}((\lambda \mathbf{x} \text{ MAX}((\lambda \mathbf{y} \text{ PAYOFF}(\mathbf{x}, \mathbf{y})), \mathbf{y}_0, \epsilon)), \mathbf{x}_0, \epsilon) \\ \mathbf{in} (\mathbf{x}^*, (\text{ARGMAX}((\lambda \mathbf{y} \text{ PAYOFF}(\mathbf{x}^*, \mathbf{y})), \mathbf{y}_0, \epsilon)))$$

Outline

- 1 Lambda Calculus
- 2 Differential Calculus in Lambda-Calculus Notation
- 3 Essence of the Derivation of Functional Reverse Mode
- 4 AD in Lambda-Calculus Notation
- 5 An Example
- 6 Benefits of this Approach**

Our Approach is Efficient

Our Approach is Efficient

- source-to-source transformation

Our Approach is Efficient

- source-to-source transformation
- no overloading
- no interpretation of tape

Our Approach is Efficient

- source-to-source transformation
- no overloading
- no interpretation of tape
- transformation conceptually done reflectively at run-time

Our Approach is Efficient

- source-to-source transformation
- no overloading
- no interpretation of tape
- transformation conceptually done reflectively at run-time
- sophisticated compilation techniques can move transformation to compile-time

Our Approach Exhibits Closure Properties

Our Approach Exhibits Closure Properties

- Can apply $\overrightarrow{\mathcal{J}}$ and $\overleftarrow{\mathcal{J}}$ to *any* function

Our Approach Exhibits Closure Properties

- Can apply $\overrightarrow{\mathcal{J}}$ and $\overleftarrow{\mathcal{J}}$ to *any* function including $\overrightarrow{\mathcal{J}}$ and $\overleftarrow{\mathcal{J}}$ themselves.

Our Approach Exhibits Closure Properties

- Can apply $\vec{\mathcal{J}}$ and $\overleftarrow{\mathcal{J}}$ to *any* function including $\vec{\mathcal{J}}$ and $\overleftarrow{\mathcal{J}}$ themselves.
- The output of $\vec{\mathcal{J}}$ and $\overleftarrow{\mathcal{J}}$ are functions.

Our Approach Exhibits Closure Properties

- Can apply $\overrightarrow{\mathcal{J}}$ and $\overleftarrow{\mathcal{J}}$ to *any* function including $\overrightarrow{\mathcal{J}}$ and $\overleftarrow{\mathcal{J}}$ themselves.
- The output of $\overrightarrow{\mathcal{J}}$ and $\overleftarrow{\mathcal{J}}$ are functions.
- Can apply $\overrightarrow{\mathcal{J}}$ and $\overleftarrow{\mathcal{J}}$ to the output of $\overrightarrow{\mathcal{J}}$ and $\overleftarrow{\mathcal{J}}$.

Our Approach Exhibits Closure Properties

- Can apply $\overrightarrow{\mathcal{J}}$ and $\overleftarrow{\mathcal{J}}$ to *any* function including $\overrightarrow{\mathcal{J}}$ and $\overleftarrow{\mathcal{J}}$ themselves.
- The output of $\overrightarrow{\mathcal{J}}$ and $\overleftarrow{\mathcal{J}}$ are functions.
- Can apply $\overrightarrow{\mathcal{J}}$ and $\overleftarrow{\mathcal{J}}$ to the output of $\overrightarrow{\mathcal{J}}$ and $\overleftarrow{\mathcal{J}}$.
- Can take derivatives of arbitrary order.

Our Approach Enhances Modularity

$$\text{ARGMIN}_1 (f, f') \quad \triangleq \quad \dots$$

Our Approach Enhances Modularity

$$\text{ARGMIN}_1 (f, f') \quad \triangleq \quad \dots$$
$$\dots \text{ARGMIN}_1 (f, f') \dots$$

Our Approach Enhances Modularity

$$\text{ARGMIN}_1 (f, f') \quad \triangleq \quad \dots$$

$$\text{ARGMIN}_2 (f, f', f'') \quad \triangleq \quad \dots$$

$$\dots \text{ARGMIN}_1 (f, f') \dots$$

Our Approach Enhances Modularity

$$\text{ARGMIN}_1 (f, f') \quad \triangleq \quad \dots$$

$$\text{ARGMIN}_2 (f, f', f'') \quad \triangleq \quad \dots$$

$$\dots \text{ARGMIN}_2 (f, f', f'') \dots$$

Our Approach Enhances Modularity

$$\text{ARGMIN}_1 f \triangleq \dots (\overleftrightarrow{\mathcal{J}} f) \dots$$

Our Approach Enhances Modularity

$$\text{ARGMIN}_1 f \triangleq \dots (\overleftrightarrow{\mathcal{J}} f) \dots$$

$$\dots \text{ARGMIN}_1 f \dots$$

Our Approach Enhances Modularity

$$\text{ARGMIN}_1 f \triangleq \dots (\overleftrightarrow{\mathcal{J}} f) \dots$$

$$\text{ARGMIN}_2 f \triangleq \dots (\overleftrightarrow{\mathcal{J}} f) \dots (\overleftrightarrow{\mathcal{J}} (\overleftrightarrow{\mathcal{J}} f)) \dots$$

$$\dots \text{ARGMIN}_1 f \dots$$

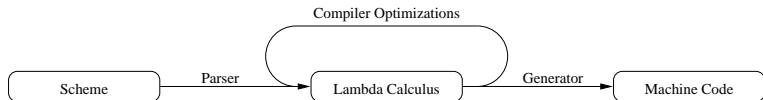
Our Approach Enhances Modularity

$$\text{ARGMIN}_1 f \triangleq \dots (\overleftrightarrow{\mathcal{J}} f) \dots$$

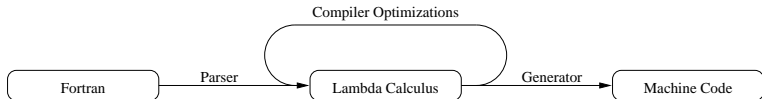
$$\text{ARGMIN}_2 f \triangleq \dots (\overleftrightarrow{\mathcal{J}} f) \dots (\overleftrightarrow{\mathcal{J}} (\overleftrightarrow{\mathcal{J}} f)) \dots$$

... ARGMIN₂ f ...

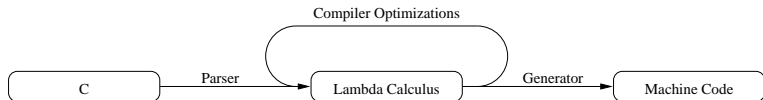
Lambda the Ultimate Intermediate Language



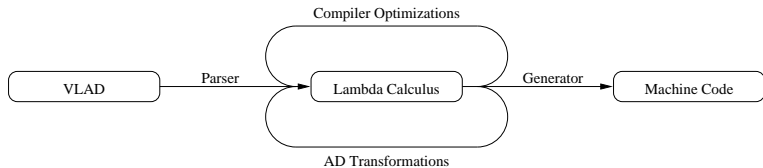
Lambda the Ultimate Intermediate Language



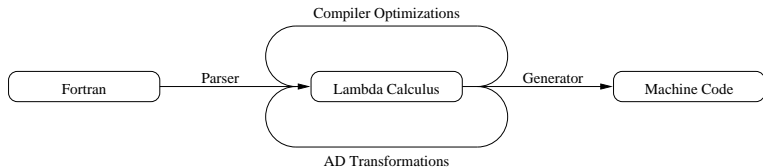
Lambda the Ultimate Intermediate Language



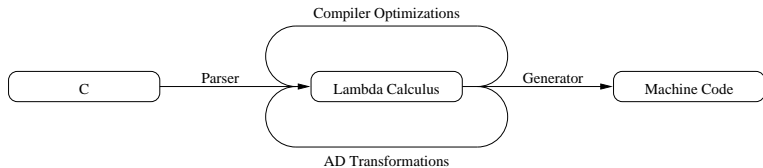
Lambda the Ultimate Intermediate Language *for AD*



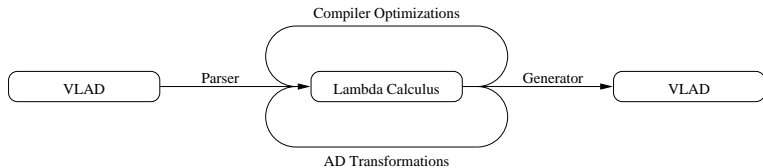
Lambda the Ultimate Intermediate Language *for AD*



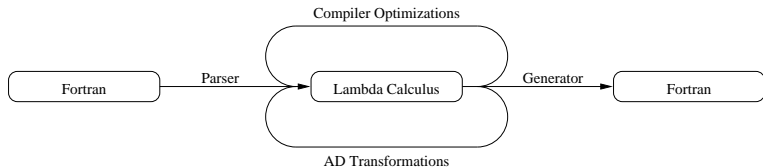
Lambda the Ultimate Intermediate Language *for AD*



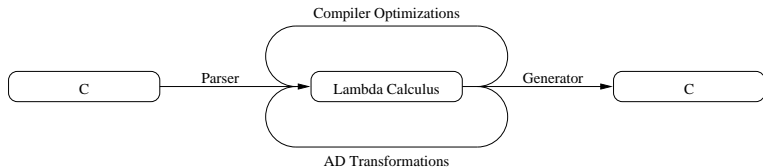
Lambda the Ultimate Intermediate Language *for AD*



Lambda the Ultimate Intermediate Language *for AD*



Lambda the Ultimate Intermediate Language *for AD*



Our Work

Our Work

- Prior Work

Our Work

- Prior Work

STALIN compiler for SCHEME

Our Work

- Prior Work

STALIN compiler for SCHEME
ruthless, brutal, good at execution

Our Work

- Prior Work

STALIN compiler for SCHEME
ruthless, brutal, good at execution
20 × FORTRAN

Our Work

- Prior Work

STALIN compiler for SCHEME
ruthless, brutal, good at execution
20 × FORTRAN

- Current Work

Our Work

- Prior Work

STALIN compiler for SCHEME
ruthless, brutal, good at execution
20 × FORTRAN

- Current Work

theory: $\lambda\nabla$ -calculus

Our Work

- Prior Work

STALIN compiler for SCHEME
ruthless, brutal, good at execution
 $20 \times$ FORTRAN

- Current Work

theory: $\lambda\nabla$ -calculus
 λ -calculus + $\overrightarrow{\mathcal{J}}$ + $\overleftarrow{\mathcal{J}}$

Our Work

- Prior Work

STALIN compiler for SCHEME
 ruthless, brutal, good at execution
 $20 \times$ FORTRAN

- Current Work

theory: $\lambda\nabla$ -calculus
 λ -calculus + $\overrightarrow{\mathcal{J}}$ + $\overleftarrow{\mathcal{J}}$
language: VLAD

Our Work

- Prior Work

STALIN compiler for SCHEME
 ruthless, brutal, good at execution
 $20 \times$ FORTRAN

- Current Work

theory: $\lambda\nabla$ -calculus
 λ -calculus + $\overrightarrow{\mathcal{J}}$ + $\overleftarrow{\mathcal{J}}$

language: VLAD
 SCHEME + $\overrightarrow{\mathcal{J}}$ + $\overleftarrow{\mathcal{J}}$

Our Work

- Prior Work

STALIN compiler for SCHEME
 ruthless, brutal, good at execution
 $20 \times$ FORTRAN

- Current Work

theory: $\lambda\nabla$ -calculus
 λ -calculus + $\overrightarrow{\mathcal{J}}$ + $\overleftarrow{\mathcal{J}}$

language: VLAD
 SCHEME + $\overrightarrow{\mathcal{J}}$ + $\overleftarrow{\mathcal{J}}$
Functional Language for AD

Our Work

- Prior Work

STALIN compiler for SCHEME
 ruthless, brutal, good at execution
 $20 \times$ FORTRAN

- Current Work

theory: $\lambda\nabla$ -calculus
 λ -calculus + $\overrightarrow{\mathcal{J}}$ + $\overleftarrow{\mathcal{J}}$
language: VLAD
 SCHEME + $\overrightarrow{\mathcal{J}}$ + $\overleftarrow{\mathcal{J}}$
Functional Language for AD
compiler: STALIN ∇