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

Outline

Higher-Order Functions

$$\begin{aligned}\textsc{Fold}\ (m, n, u, b, i) \stackrel{\triangle}{=} &\ \mathbf{if}\ \ m > n\\ &\ \mathbf{then}\ \ i\\ &\ \mathbf{else}\ \ b\ ((u\ m), (\textsc{Fold}\ (m+1, n, u, b, i)))\end{aligned}$$

Higher-Order Functions

$$\textsc{Fold}\ (m, n, u, b, i) \stackrel{\triangle}{=} \begin{array}{l} \textbf{if}\ \ m > n \\ \textbf{then}\ \ i \\ \textbf{else}\ \ b\ ((u\ m), (\textsc{Fold}\ (m+1, n, u, b, i))) \end{array}$$

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

Higher-Order Functions

$$\textsc{Fold}\ (m, n, u, b, i) \stackrel{\triangle}{=} \begin{array}{l}\textbf{if}\ \ m > n\\ \textbf{then}\ \ i\\ \textbf{else}\ \ b\ ((u\ m), (\textsc{Fold}\ (m+1, n, u, b, i)))\end{array}$$

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

Higher-Order Functions

$$
\begin{aligned}
\textsc{Fold}\ (m, n, u, b, i) \stackrel{\triangle}{=}\ & \textbf{if}\ \ m > n \\
& \textbf{then}\ \ i \\
& \textbf{else}\ \ b\ ((u\ m), (\textsc{Fold}\ (m+1, n, u, b, i)))
\end{aligned}
$$

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

Higher-Order Functions

$$\textsc{Fold}\ (m, n, u, b, i) \stackrel{\triangle}{=} \begin{array}{l}\textbf{if}\ \ m > n\\ \textbf{then}\ \ i\\ \textbf{else}\ \ b\ ((u\ m), (\textsc{Fold}\ (m+1, n, u, b, i)))\end{array}$$

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

Lambda Expressions

Anonymous Functions

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

Lambda Expressions

Anonymous Functions

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

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

Lambda Expressions

Anonymous Functions

$$\sum_{i=m}^{n} i^2 = \textsc{Fold}\,(m, n, \textsc{sqr}, +, 0)$$

$$\textsc{sqr}\ i \triangleq i \times i$$

Lambda Expressions

Anonymous Functions

$$\sum_{i=m}^{n} i^2 \quad = \quad \textsc{Fold}\,(m, n, (\lambda i\; i \times i), +, 0)$$

Nesting, Free Variables, and Closures

$$(\lambda x\, 2 \times x)\; 3 \quad = \quad 6$$

Nesting, Free Variables, and Closures

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

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

Nesting, Free Variables, and Closures

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

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

Nesting, Free Variables, and Closures

$$
\begin{aligned}
(\lambda x\, 2 \times x)\; 3 &= 6 \\
(\lambda x\, \lambda y\, x + y)\; 3 &= \langle \{x \mapsto 3\}, \lambda y\, x + y \rangle
\end{aligned}
$$

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.

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

Outline

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Partial Derivatives

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

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

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

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

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

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

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

Jacobians

$$
\begin{array}{rcl}
f & : & \mathbb{R}^n \rightarrow \mathbb{R}^m \\
\mathbf{f} & : & (\mathbb{R}^n \rightarrow \mathbb{R})^m \\
(\mathcal{J}\ f\ \mathbf{x})[i,j] & = & (\nabla\ (\mathbf{f}[i]))[j] \\
\mathcal{J} & : & (\mathbb{R}^n \rightarrow \mathbb{R}^m) \rightarrow (\mathbb{R}^n \rightarrow \mathbb{R}^{m\times n})
\end{array}
$$

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

Reverse Mode on Imperative Programs

$$
\begin{array}{rcl}
 & \vdots & \\
x_2 & := & u\ x_1 \\
 & & \\
x_4 & := & b\ (x_2, x_3) \\
 & \vdots &
\end{array}
$$

Reverse Mode on Imperative Programs

$$
\left.\begin{array}{rcl}
 & \vdots & \\
x_2 & := & u\ x_1 \\
x_4 & := & b\ (x_2, x_3) \\
 & \vdots &
\end{array}\right\}\ \textit{forward phase}
$$

$$
\left.\begin{array}{rcl}
 & \vdots & \\
\grave{x}_2 & +:= & (\mathcal{D}_1\ b\ (x_2, x_3)) \times \grave{x}_4 \\
\grave{x}_3 & +:= & (\mathcal{D}_2\ b\ (x_2, x_3)) \times \grave{x}_4 \\
\grave{x}_1 & +:= & (\mathcal{D}\ u\ x_1) \times \grave{x}_2 \\
 & \vdots &
\end{array}\right\}\ \textit{reverse phase}
$$

Reverse Mode on Imperative Programs

$$
\left.\begin{array}{lcl}
 & \vdots & \\
x_2 & := & u\ x_1 \\
x_4 & := & b\ (x_2, x_3) \\
 & \vdots &
\end{array}\right\}\ \textit{forward phase}
$$

$$
\left.\begin{array}{lcl}
 & \vdots & \\
\grave{x}_2 & +:= & (\mathcal{D}_1\ b\ (x_2, x_3)) \times \grave{x}_4 \\
\grave{x}_3 & +:= & (\mathcal{D}_2\ b\ (x_2, x_3)) \times \grave{x}_4 \\
\grave{x}_1 & +:= & (\mathcal{D}\ u\ x_1) \times \grave{x}_2 \\
 & \vdots &
\end{array}\right\}\ \textit{reverse phase}
$$

Reverse Mode on Imperative Programs

$$
\left.
\begin{array}{lcl}
 & \vdots & \\
x_2 & := & u\ x_1 \\
x_4 & := & b\ (x_2, x_3) \\
 & \vdots &
\end{array}
\right\}\ \textit{forward phase}
$$

$$
\left.
\begin{array}{lcl}
 & \vdots & \\
\overleftarrow{x_2} & +:= & (\mathcal{D}_1\ b\ (x_2, x_3)) \times \overleftarrow{x_4} \\
\overleftarrow{x_3} & +:= & (\mathcal{D}_2\ b\ (x_2, x_3)) \times \overleftarrow{x_4} \\
\overleftarrow{x_1} & +:= & (\mathcal{D}\ u\ x_1) \times \overleftarrow{x_2} \\
 & \vdots &
\end{array}
\right\}\ \textit{reverse phase}
$$

Reverse Mode on Imperative Programs

$$
\left.\begin{array}{rcl}
 & \vdots & \\
x_2 & := & u\ x_1 \\
x_4 & := & b\ (x_2, x_3) \\
 & \vdots &
\end{array}\right\}\ \textit{forward phase}
$$

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

Reverse Mode on Imperative Programs

$$
\left.\begin{array}{rcl}
 & \vdots & \\
x_2 & := & u\ x_1 \\
 & & \\
{\color{red}x_2} & := & b\ (x_2, x_3) \\
 & \vdots &
\end{array}\right\}\ \textit{forward phase}
$$

$$
\left.\begin{array}{rcl}
 & \vdots & \\
\grave{x}_2 & +:= & (\mathcal{D}_1\ b\ (x_2, x_3)) \times \grave{x}_2 \\
\grave{x}_3 & +:= & (\mathcal{D}_2\ b\ (x_2, x_3)) \times \grave{x}_2 \\
 & & \\
\grave{x}_1 & +:= & (\mathcal{D}\ u\ x_1) \times \grave{x}_2 \\
 & \vdots &
\end{array}\right\}\ \textit{reverse phase}
$$

Reverse Mode on Imperative Programs

$$
\left.
\begin{array}{rcl}
 & \vdots & \\
x_2 & := & u\ x_1 \\
 & & \\
x_2 & := & b\ (x_2, x_3) \\
 & \vdots &
\end{array}
\right\}\ \textit{forward phase}
$$

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

Reverse Mode on Imperative Programs

$$
\left.\begin{array}{lll}
 & \vdots & \\
x_2 & := & u\ x_1 \\
\text{PUSH } x_2 & & \\
x_2 & := & b\ (x_2, x_3) \\
 & \vdots &
\end{array}\right\}\ \textit{forward phase}
$$

$$
\left.\begin{array}{lll}
 & \vdots & \\
\text{POP } x_2 & & \\
\grave{x}_2 & +:= & (\mathcal{D}_1\ b\ (x_2, x_3)) \times \grave{x}_2 \\
\grave{x}_3 & +:= & (\mathcal{D}_2\ b\ (x_2, x_3)) \times \grave{x}_2 \\
\grave{x}_1 & +:= & (\mathcal{D}\ u\ x_1) \times \grave{x}_2 \\
 & \vdots &
\end{array}\right\}\ \textit{reverse phase}
$$

Reverse Mode on Imperative Programs

$$
\left.
\begin{array}{lcl}
 & \vdots & \\
\text{PUSH } x_1 & & \\
x_2 & := & u\ x_1 \\
\text{PUSH } x_2 & & \\
\text{PUSH } x_3 & & \\
x_2 & := & b\ (x_2, x_3) \\
 & \vdots &
\end{array}
\right\}\ \textit{forward phase}
$$

$$
\left.
\begin{array}{lcl}
 & \vdots & \\
\text{POP } x_3 & & \\
\text{POP } x_2 & & \\
\overleftarrow{x_2} & +:= & (\mathcal{D}_1\ b\ (x_2, x_3)) \times \overleftarrow{x_2} \\
\overleftarrow{x_3} & +:= & (\mathcal{D}_2\ b\ (x_2, x_3)) \times \overleftarrow{x_2} \\
\text{POP } x_1 & & \\
\overleftarrow{x_1} & +:= & (\mathcal{D}\ u\ x_1) \times \overleftarrow{x_2} \\
 & \vdots &
\end{array}
\right\}\ \textit{reverse phase}
$$

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

Subroutines and Tapes

Unary Primitives

$$u : x \mapsto y$$

Subroutines and Tapes

Unary Primitives

$$u : x \mapsto y$$

$$\overleftarrow{u} : x \mapsto y \quad \triangleq \quad \left\{ \begin{array}{lll} \text{PUSH } x & & \\ y & := & u\,x \end{array} \right.$$

$$\bar{u} : \overleftharpoon{y} \mapsto \overleftharpoon{x} \quad \triangleq \quad \left\{ \begin{array}{lll} \text{POP } x & & \\ \overleftharpoon{x} & +:= & (\mathcal{D}\,u\,x) \times \overleftharpoon{y} \end{array} \right.$$

Subroutines and Tapes

Binary Primitives

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

Subroutines and Tapes

Binary Primitives

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

$$\overleftharpoon{b} : (x, y) \mapsto z \quad \triangleq \quad \left\{ \begin{array}{lll} \text{PUSH } x & & \\ \text{PUSH } y & & \\ z & := & b\ (x, y) \end{array} \right.$$

$$\overline{b} : \grave{\overline{z}} \mapsto (\grave{\overline{x}}, \grave{\overline{y}}) \quad \triangleq \quad \left\{ \begin{array}{lll} \text{POP } x & & \\ \text{POP } y & & \\ \grave{\overline{x}} & +:= & (\mathcal{D}_1\ b\ (x, y)) \times \grave{\overline{z}} \\ \grave{\overline{y}} & +:= & (\mathcal{D}_2\ b\ (x, y)) \times \grave{\overline{z}} \end{array} \right.$$

Subroutines and Tapes

User-Defined Functions

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

Subroutines and Tapes

User-Defined Functions

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

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

$$\bar{f} : \grave{\overline{\mathbf{y}}} \mapsto \grave{\overline{\mathbf{x}}} \quad \triangleq \quad \left\{ \begin{array}{lcl} \grave{\overline{\mathbf{x}_n}} & +:= & \bar{f_n}\ \grave{\overline{\mathbf{y}_n}} \\ & \vdots & \\ \grave{\overline{\mathbf{x}_1}} & +:= & \bar{f_1}\ \grave{\overline{\mathbf{y}_1}} \end{array} \right.$$

Representing the Tape as Function Arguments and Results

Unary Primitives

$$u : x \mapsto y$$

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

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

Representing the Tape as Function Arguments and Results

Binary Primitives

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

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

$$\overline{b} : ((x, y), \overleftharpoondown{z}) \mapsto (\overleftharpoondown{x}, \overleftharpoondown{y}) \quad \triangleq \quad \left\{ \begin{array}{lll} \overleftharpoondown{x} & +:= & (\mathcal{D}_1\ b\ (x, y)) \times \overleftharpoondown{z} \\ \overleftharpoondown{y} & +:= & (\mathcal{D}_2\ b\ (x, y)) \times \overleftharpoondown{z} \end{array} \right.$$

Representing the Tape as Function Arguments and Results

User-Defined Functions

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

$$\overleftharpoon{f} : \mathbf{x} \mapsto (\mathbf{y}, (\mathbf{t}_1, \ldots, \mathbf{t}_n)) \quad \triangleq \quad \left\{ \begin{array}{lcl} \mathbf{y}_1, \mathbf{t}_1 & := & \overleftharpoon{f_1}\ \mathbf{x}_1 \\ & \vdots & \\ \mathbf{y}_n, \mathbf{t}_n & := & \overleftharpoon{f_n}\ \mathbf{x}_n \end{array} \right.$$

$$\bar{f} : ((\mathbf{t}_1, \ldots, \mathbf{t}_n), \grave{\mathbf{y}}) \mapsto \grave{\mathbf{x}} \quad \triangleq \quad \left\{ \begin{array}{lcl} \grave{\mathbf{x}}_n & +:= & \bar{f_n}\ (\mathbf{t}_n, \grave{\mathbf{y}}_n) \\ & \vdots & \\ \grave{\mathbf{x}}_1 & +:= & \bar{f_1}\ (\mathbf{t}_1, \grave{\mathbf{y}}_1) \end{array} \right.$$

Representing the Tape as Closures

Unary Primitives

$$u : x \mapsto y$$

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

Representing the Tape as Closures

Binary Primitives

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

$$\overleftarrow{b} : (x, y) \mapsto (z, \overline{b}) \triangleq \left\{ \begin{array}{lll} z & := & b\ (x, y) \\ \overline{b} : \grave{z} \mapsto (\grave{x}, \grave{y}) & \triangleq & \left\{ \begin{array}{lll} \grave{x} & +:= & (\mathcal{D}_1\ b\ (x, y)) \times \grave{z} \\ \grave{y} & +:= & (\mathcal{D}_2\ b\ (x, y)) \times \grave{z} \end{array} \right. \end{array} \right.$$

Representing the Tape as Closures

User-Defined Functions

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

$$\overleftarrow{f} : \mathbf{x} \mapsto (\mathbf{y}, \bar{f}) \quad \triangleq \quad \left\{ \begin{array}{lll} \mathbf{y}_1, \bar{f}_1 & := & \overleftarrow{f_1}\ \mathbf{x}_1 \\ & \vdots & \\ \mathbf{y}_n, \bar{f}_n & := & \overleftarrow{f_n}\ \mathbf{x}_n \\ \bar{f} : \grave{\mathbf{y}} \mapsto \grave{\mathbf{x}} & \triangleq & \left\{ \begin{array}{lll} \grave{\mathbf{x}}_n & +:= & \bar{f}_n\ \grave{\mathbf{y}}_n \\ & \vdots & \\ \grave{\mathbf{x}}_1 & +:= & \bar{f}_1\ \grave{\mathbf{y}}_1 \end{array} \right. \end{array} \right.$$

Details for Handling Closures Omitted

Outline

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

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

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

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

$$\overline{\tau_1 \xrightarrow{\tau_1',\ldots,\tau_n'} \tau_2} = \overline{\tau_1'} \times \cdots \times \overline{\tau_n'}$$

New Formulation of AD as Higher-Order Functions

Forward Types

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

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

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

$$\overrightharpoondown{\tau_1 \xrightarrow{\tau_1', \ldots, \tau_n'} \tau_2} = \overrightharpoondown{\tau_1} \xrightarrow{\overrightharpoondown{\tau_1'}, \ldots, \overrightharpoondown{\tau_n'}} \overrightharpoondown{\tau_2}$$

New Formulation of AD as Higher-Order Functions

Sensitivity Types

$$
\begin{aligned}
\overleftarrow{\mathbf{null}} &= \mathbf{null}\\[1em]
\overleftarrow{\mathbb{R}} &= \mathbb{R}\\[1em]
\overleftarrow{\tau_1 \times \tau_2} &= \overleftarrow{\tau_1} \times \overleftarrow{\tau_2}\\[1em]
\overleftarrow{\tau_1 \xrightarrow{\tau_1',\ldots,\tau_n'} \tau_2} &= \overleftarrow{\tau_1'} \times \cdots \times \overleftarrow{\tau_n'}
\end{aligned}
$$

New Formulation of AD as Higher-Order Functions

Reverse Types

$$\overleftarrow{\textbf{null}} = \textbf{null}$$

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

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

$$\overleftarrow{\tau_1 \xrightarrow{\tau_1', \ldots, \tau_n'} \tau_2} = \overleftarrow{\tau_1} \xrightarrow{\overleftarrow{\tau_1'}, \ldots, \overleftarrow{\tau_n'}} (\overleftarrow{\tau_2} \times (\grave{\overline{\tau_2}} \to (\grave{\overline{\tau_1'}} \times \cdots \times \grave{\overline{\tau_n'}}) \times \grave{\overline{\tau_1}}))$$

New Formulation of AD as Higher-Order Functions

Forward Mode: $\overrightarrow{\mathcal{J}} : \tau \rightarrow \overrightharpoon{\tau}$
Reverse Mode: $\overleftarrow{\mathcal{J}} : \tau \rightarrow \overleftharpoon{\tau}$

Outline

Saddle Points

Continuous Two-Person Zero Sum Games

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

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

$\textsc{Payoff} : \mathbb{R}^m \times \mathbb{R}^n \rightarrow \mathbb{R}$

$\min_{\mathbf{x}} \max_{\mathbf{y}} \textsc{Payoff}\,(\mathbf{x}, \mathbf{y})$

Saddle Points

Continuous Two-Person Zero Sum Games

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

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

$\textsc{Payoff} : \mathbb{R}^m \times \mathbb{R}^n \to \mathbb{R}$

$\min_{\mathbf{x}} \max_{\mathbf{y}} \textsc{Payoff}\ (\mathbf{x}, \mathbf{y})$

$$\begin{aligned}(\mathbf{x}^*, \mathbf{y}^*) = \mathbf{let}\ \mathbf{x}^* &\triangleq \textsc{Argmin}\ ((\lambda \mathbf{x}\ \textsc{Max}\ ((\lambda \mathbf{y}\ \textsc{Payoff}\ (\mathbf{x}, \mathbf{y})), \mathbf{y}_0, \epsilon)), \mathbf{x}_0, \epsilon)\\ \mathbf{in}\ &(\mathbf{x}^*, (\textsc{Argmax}\ ((\lambda \mathbf{y}\ \textsc{Payoff}\ (\mathbf{x}^*, \mathbf{y})), \mathbf{y}_0, \epsilon)))\end{aligned}$$

Outline

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 $\overrightarrow{\mathcal{J}}$ 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.

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

$$\textsc{Argmin}_1\ (f, f') \quad \triangleq \quad \ldots$$

Our Approach Enhances Modularity

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

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

Our Approach Enhances Modularity

$$
\begin{aligned}
\textsc{Argmin}_1\ (f, f') &\triangleq \ldots \\
\textsc{Argmin}_2\ (f, f', f'') &\triangleq \ldots
\end{aligned}
$$

$$\ldots \textsc{Argmin}_1\ (f, f') \ldots$$

Our Approach Enhances Modularity

$$\begin{aligned}\textsc{Argmin}_1\ (f,f') &\stackrel{\triangle}{=} \ldots\\ \textsc{Argmin}_2\ (f,f',f'') &\stackrel{\triangle}{=} \ldots\end{aligned}$$

$$\ldots\textsc{Argmin}_2\ (f,f',f'')\ldots$$

Our Approach Enhances Modularity

$$\textsc{Argmin}_1\, f \;\triangleq\; \ldots(\overleftrightarrow{\mathcal{J}}\, f)\ldots$$

Our Approach Enhances Modularity

$$\textsc{Argmin}_1\, f \;\triangleq\; \ldots (\overleftrightarrow{\mathcal{J}}\, f) \ldots$$

$$\ldots \textsc{Argmin}_1\, f \ldots$$

Our Approach Enhances Modularity

$$\textsc{Argmin}_1\, f \triangleq \ldots(\overleftrightarrow{\mathcal{J}}\, f)\ldots$$

$$\textsc{Argmin}_2\, f \triangleq \ldots(\overleftrightarrow{\mathcal{J}}\, f)\ldots(\overleftrightarrow{\mathcal{J}}\, (\overleftrightarrow{\mathcal{J}}\, f))\ldots$$

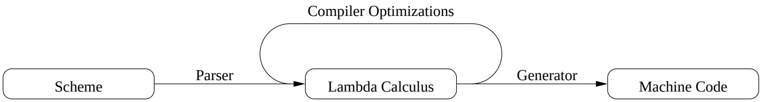
$$\ldots\textsc{Argmin}_1\, f\ldots$$

Our Approach Enhances Modularity

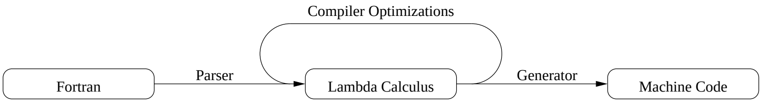
$$\begin{aligned}
\text{ARGMIN}_1\, f &\stackrel{\triangle}{=} \ldots(\overleftrightarrow{\mathcal{J}}\, f)\ldots \\
\text{ARGMIN}_2\, f &\stackrel{\triangle}{=} \ldots(\overleftrightarrow{\mathcal{J}}\, f)\ldots(\overleftrightarrow{\mathcal{J}}\ (\overleftrightarrow{\mathcal{J}}\, f))\ldots
\end{aligned}$$

$$\ldots \text{ARGMIN}_2\, f \ldots$$

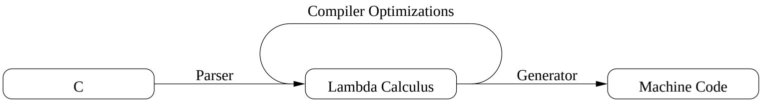
Lambda the Ultimate Intermediate Language



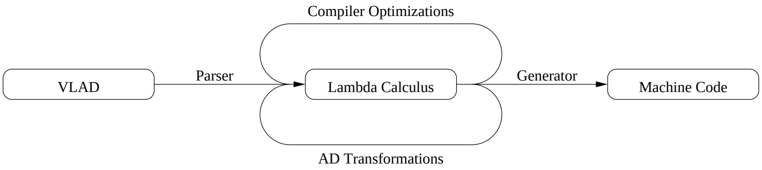
Lambda the Ultimate Intermediate Language



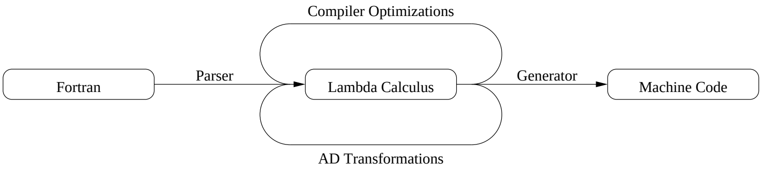
Lambda the Ultimate Intermediate Language



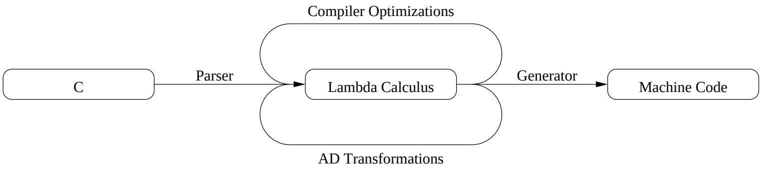
Lambda the Ultimate Intermediate Language *for AD*



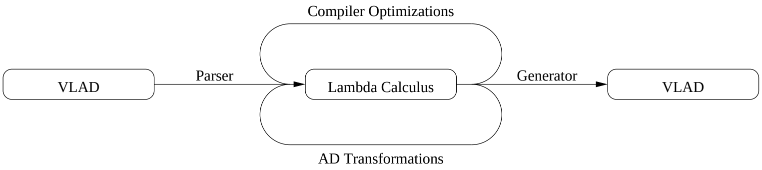
Lambda the Ultimate Intermediate Language *for AD*



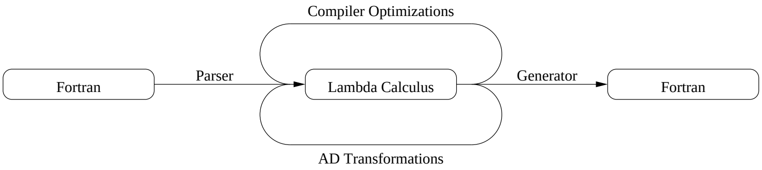
Lambda the Ultimate Intermediate Language *for AD*



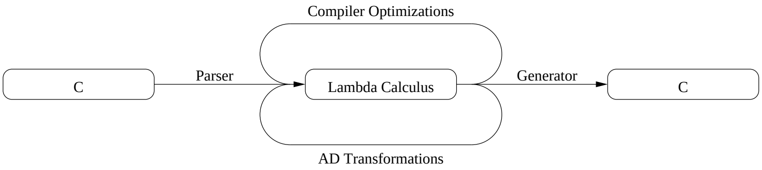
Lambda the Ultimate Intermediate Language *for AD*



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
	$20 \times$ FORTRAN

Our Work

- Prior Work

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

- Current Work

Our Work

- Prior Work

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