Automatic Differentiation of Functional Programs and its use for Probabilistic Programming

Jeffrey Mark Siskind
qobi@purdue.edu

School of Electrical and Computer Engineering
Purdue University

NJPLS
Princeton University
3 April 2009

Part of this talk covers joint work with Barak A. Pearlmutter.

The Essence

(define (f x) $2x^3$)

The Essence

`(define` $(f\ \ x)\ \ 2x^3$`)` $\quad\rightsquigarrow\quad$ `(define` $(f'\ \ x)\ \ 6x^2$`)`

The Essence

```
(define (g x) sin f(x))
```

$$\texttt{(define (}g\ x\texttt{)}\ \sin f(x)\texttt{)}$$

The Essence

$$\texttt{(define (}g\ x\texttt{)}\ \sin f(x)\texttt{)} \rightsquigarrow \texttt{(define (}g'\ x\texttt{)}\ f'(x)\cos f(x)\texttt{)}$$

The Essence

$$\texttt{(define (}g\ x\texttt{)}\ \sin f(x)\texttt{)} \rightsquigarrow \texttt{(define (}g'\ x\texttt{)}\ f'(x)\cos f(x)\texttt{)}$$

The Essence

(define (f x) $2x^3$)

(define (g x) $\sin f(x)$) $\rightsquigarrow$ (define (g' x) $f'(x)\cos f(x)$)

The Essence

`(define` $(f\ x)$ $2x^3$`)`

`(define` $(g\ x)$ $\sin f(x)$`)` $\rightsquigarrow$ `(define` $(g'\ x)$ $f'(x)\cos f(x)$`)`

The Essence

`(define (`f` `x`) `$2x^3$`)`

`(define (`g` `x`) `$\sin f(x)$`)` $\rightsquigarrow$ `(define (`g'` `x`) `$f'(x)\cos f(x)$`)`

The Essence

`(define (`f` `x`) `$2x^3$`)` $\rightsquigarrow$ `(define (`f'` `x`) `$6x^2$`)`

`(define (`g` `x`) `$\sin f(x)$`)` $\rightsquigarrow$ `(define (`g'` `x`) `$f'(x)\cos f(x)$`)`

The Essence

`(define (`f x`)` $2x^3$`)` $\rightsquigarrow$ `(define (`f' x`)` $6x^2$`)`

`(define (`g x`)` $\sin f(x)$`)` $\rightsquigarrow$ `(define (`g' x`)` $f'(x) \cos f(x)$`)`

`(`$\mathcal{D}$ g`)`

The Essence

(define $(f\ \ x)$ $2x^3$) $\rightsquigarrow$ (define $(f'\ \ x)$ $6x^2$)

(define $(g\ \ x)$ $\sin f(x)$) $\rightsquigarrow$ (define $(g'\ \ x)$ $f'(x)\cos f(x)$)

($\mathcal{D}$ g)

The Essence

$\texttt{(define } (f\ \ x)\ \ 2x^3\texttt{)} \quad\rightsquigarrow\quad \texttt{(define } (f'\ \ x)\ \ 6x^2\texttt{)}$

$\texttt{(define } (g\ \ x)\ \ \sin f(x)\texttt{)} \quad\rightsquigarrow\quad \texttt{(define } (g'\ \ x)\ \ f'(x)\cos f(x)\texttt{)}$

$\texttt{(}\mathcal{D}\ \ g\texttt{)} \qquad \langle\{f \mapsto \lambda x\ 2x^3\}, \lambda x\ \sin f(x)\rangle$

The Essence

$$\texttt{(define (}f\ x\texttt{)}\ 2x^3\texttt{)} \rightsquigarrow \texttt{(define (}f'\ x\texttt{)}\ 6x^2\texttt{)}$$

$$\texttt{(define (}g\ x\texttt{)}\ \sin f(x)\texttt{)} \rightsquigarrow \texttt{(define (}g'\ x\texttt{)}\ f'(x)\cos f(x)\texttt{)}$$

$$\texttt{(}\mathcal{D}\ g\texttt{)} \qquad \langle\{f \mapsto \lambda x\ 2x^3\}, \lambda x\ \sin f(x)\rangle$$

The Essence

$\texttt{(define (}f\ \ x\texttt{)}\ \ 2x^3\texttt{)} \quad \rightsquigarrow \quad \texttt{(define (}f'\ \ x\texttt{)}\ \ 6x^2\texttt{)}$

$\texttt{(define (}g\ \ x\texttt{)}\ \ \sin f(x)\texttt{)} \quad \rightsquigarrow \quad \texttt{(define (}g'\ \ x\texttt{)}\ \ f'(x)\cos f(x)\texttt{)}$

$\texttt{(}\mathcal{D}\ \ g\texttt{)} \qquad \langle\{f \mapsto \lambda x\, 2x^3\}, \lambda x\ \sin f(x)\rangle$

The Essence

$$\texttt{(define (}f\ x\texttt{)}\ 2x^3\texttt{)} \rightsquigarrow \texttt{(define (}f'\ x\texttt{)}\ 6x^2\texttt{)}$$

$$\texttt{(define (}g\ x\texttt{)}\ \sin f(x)\texttt{)} \rightsquigarrow \texttt{(define (}g'\ x\texttt{)}\ f'(x)\cos f(x)\texttt{)}$$

$$\texttt{(}\mathcal{D}\ g\texttt{)} \Longrightarrow \texttt{(}\mathcal{D}\ \langle\{f \mapsto \lambda x\ 2x^3\}, \lambda x\ \sin f(x)\rangle\texttt{)}$$

The Essence

$$\texttt{(define (}f\ x\texttt{)}\ 2x^3\texttt{)} \rightsquigarrow \texttt{(define (}f'\ x\texttt{)}\ 6x^2\texttt{)}$$

$$\texttt{(define (}g\ x\texttt{)}\ \sin f(x)\texttt{)} \rightsquigarrow \texttt{(define (}g'\ x\texttt{)}\ f'(x)\cos f(x)\texttt{)}$$

$$\texttt{(}\mathcal{D}\ g\texttt{)} \Longrightarrow \texttt{(}\mathcal{D}\ \langle\{f \mapsto \lambda x\, 2x^3\}, \lambda x\ \sin f(x)\rangle\texttt{)}$$

$$\Longrightarrow \langle\{f \mapsto \lambda x\, 2x^3, f' \mapsto \lambda x\, 6x^2\}, \lambda x\, f'(x)\cos f(x)\rangle$$

The Essence

$$\texttt{(define (}f\ x\texttt{)}\ 2x^3\texttt{)} \quad \rightsquigarrow \quad \texttt{(define (}f'\ x\texttt{)}\ 6x^2\texttt{)}$$

$$\texttt{(define (}g\ x\texttt{)}\ \sin f(x)\texttt{)} \quad \rightsquigarrow \quad \texttt{(define (}g'\ x\texttt{)}\ f'(x)\cos f(x)\texttt{)}$$

$$\texttt{(}\mathcal{D}\ g\texttt{)} \quad \Longrightarrow \quad \texttt{(}\mathcal{D}\ \langle\{f \mapsto \lambda x\, 2x^3\}, \lambda x\ \sin f(x)\rangle\texttt{)}$$

$$\Longrightarrow \langle\{f \mapsto \lambda x\, 2x^3, f' \mapsto \lambda x\, 6x^2\}, \lambda x\, f'(x)\cos f(x)\rangle$$

The Essence

$$\texttt{(define (}f\ x\texttt{)}\ 2x^3\texttt{)} \rightsquigarrow \texttt{(define (}f'\ x\texttt{)}\ 6x^2\texttt{)}$$

$$\texttt{(define (}g\ x\texttt{)}\ \sin f(x)\texttt{)} \rightsquigarrow \texttt{(define (}g'\ x\texttt{)}\ f'(x)\cos f(x)\texttt{)}$$

$$\texttt{(}\mathcal{D}\ g\texttt{)} \Longrightarrow \texttt{(}\mathcal{D}\ \langle\{f \mapsto \lambda x\ 2x^3\}, \lambda x\ \sin f(x)\rangle\texttt{)}$$

$$\Longrightarrow \langle\{f \mapsto \lambda x\ 2x^3, f' \mapsto \lambda x\ 6x^2\}, \lambda x\ f'(x)\cos f(x)\rangle$$

The Essence

$$\texttt{(define (}f\ x\texttt{)}\ 2x^3\texttt{)} \rightsquigarrow \texttt{(define (}f'\ x\texttt{)}\ 6x^2\texttt{)}$$

$$\texttt{(define (}g\ x\texttt{)}\ \sin f(x)\texttt{)} \rightsquigarrow \texttt{(define (}g'\ x\texttt{)}\ f'(x)\cos f(x)\texttt{)}$$

$$\texttt{(}\mathcal{D}\ g\texttt{)} \Longrightarrow \texttt{(}\mathcal{D}\ \langle\{f \mapsto \lambda x\ 2x^3\}, \lambda x\ \sin f(x)\rangle\texttt{)}$$

$$\Longrightarrow \langle\{f \mapsto \lambda x\ 2x^3, f' \mapsto \lambda x\ 6x^2\}, \lambda x\ f'(x)\cos f(x)\rangle$$

The Essence

$$\texttt{(define (}f\ \ x\texttt{)}\ 2x^3\texttt{)} \rightsquigarrow \texttt{(define (}f'\ \ x\texttt{)}\ 6x^2\texttt{)}$$

$$\texttt{(define (}g\ \ x\texttt{)}\ \sin f(x)\texttt{)} \rightsquigarrow \texttt{(define (}g'\ \ x\texttt{)}\ f'(x)\cos f(x)\texttt{)}$$

$$\texttt{(}\mathcal{D}\ \ g\texttt{)} \Longrightarrow \texttt{(}\mathcal{D}\ \ \langle\{f \mapsto \lambda x\ 2x^3\}, \lambda x\ \sin f(x)\rangle\texttt{)}$$

$$\Longrightarrow \langle\{f \mapsto \lambda x\ 2x^3, f' \mapsto \lambda x\ 6x^2\}, \lambda x\, f'(x)\cos f(x)\rangle$$

$$\texttt{(map-closure}\ f\ \ \langle\{x_1 \mapsto v_1, \ldots\}, e\rangle\texttt{)} \Longrightarrow \langle\{x_1 \mapsto f(v_1), \ldots\}, e\rangle$$

The Essence

$$\texttt{(define (}f\ x\texttt{)}\ 2x^3\texttt{)} \rightsquigarrow \texttt{(define (}f'\ x\texttt{)}\ 6x^2\texttt{)}$$

$$\texttt{(define (}g\ x\texttt{)}\ \sin f(x)\texttt{)} \rightsquigarrow \texttt{(define (}g'\ x\texttt{)}\ f'(x)\cos f(x)\texttt{)}$$

$$\texttt{(}\mathcal{D}\ g\texttt{)} \Longrightarrow \texttt{(}\mathcal{D}\ \langle\{f \mapsto \lambda x\, 2x^3\}, \lambda x\ \sin f(x)\rangle\texttt{)}$$

$$\Longrightarrow \langle\{f \mapsto \lambda x\, 2x^3, f' \mapsto \lambda x\, 6x^2\}, \lambda x\, f'(x)\cos f(x)\rangle$$

$$\texttt{(map-closure}\ f\ \langle\{x_1 \mapsto v_1, \ldots\}, e\rangle\texttt{)} \Longrightarrow \langle\{x_1 \mapsto f(v_1), \ldots\}, e\rangle$$

The Essence

$$\texttt{(define (}f\ x\texttt{)}\ 2x^3\texttt{)} \rightsquigarrow \texttt{(define (}f'\ x\texttt{)}\ 6x^2\texttt{)}$$

$$\texttt{(define (}g\ x\texttt{)}\ \sin f(x)\texttt{)} \rightsquigarrow \texttt{(define (}g'\ x\texttt{)}\ f'(x)\cos f(x)\texttt{)}$$

$$\texttt{(}\mathcal{D}\ g\texttt{)} \Longrightarrow \texttt{(}\mathcal{D}\ \langle\{f \mapsto \lambda x\ 2x^3\}, \lambda x\ \sin f(x)\rangle\texttt{)}$$

$$\Longrightarrow \langle\{f \mapsto \lambda x\ 2x^3, f' \mapsto \lambda x\ 6x^2\}, \lambda x\ f'(x)\cos f(x)\rangle$$

$$\texttt{(map-closure}\ f\ \langle\{x_1 \mapsto v_1, \ldots\}, e\rangle\texttt{)} \Longrightarrow \langle\{x_1 \mapsto f(v_1), \ldots\}, e\rangle$$

need reflective transformation of closure bodies

The Essence

$$\texttt{(define (}f\ x\texttt{)}\ 2x^3\texttt{)} \rightsquigarrow \texttt{(define (}f'\ x\texttt{)}\ 6x^2\texttt{)}$$

$$\texttt{(define (}g\ x\texttt{)}\ \sin f(x)\texttt{)} \rightsquigarrow \texttt{(define (}g'\ x\texttt{)}\ f'(x)\cos f(x)\texttt{)}$$

$$\texttt{(}\mathcal{D}\ g\texttt{)} \Longrightarrow \texttt{(}\mathcal{D}\ \langle\{f \mapsto \lambda x\ 2x^3\}, \lambda x\ \sin f(x)\rangle\texttt{)}$$

$$\Longrightarrow \langle\{f \mapsto \lambda x\ 2x^3, f' \mapsto \lambda x\ 6x^2\}, \lambda x\, f'(x)\cos f(x)\rangle$$

$$\texttt{(map-closure}\ f\ \langle\{x_1 \mapsto v_1, \ldots\}, e\rangle\texttt{)} \Longrightarrow \langle\{x_1 \mapsto f(v_1), \ldots\}, e\rangle$$

need reflective transformation of closure bodies
want transformation done at compile time

The Essence

$\texttt{(define (}f\ x\texttt{)}\ 2x^3\texttt{)} \rightsquigarrow \texttt{(define (}f'\ x\texttt{)}\ 6x^2\texttt{)}$

$\texttt{(define (}g\ x\texttt{)}\ \sin f(x)\texttt{)} \rightsquigarrow \texttt{(define (}g'\ x\texttt{)}\ f'(x)\cos f(x)\texttt{)}$

$\texttt{(}\mathcal{D}\ g\texttt{)} \Longrightarrow \texttt{(}\mathcal{D}\ \langle\{f \mapsto \lambda x\ 2x^3\}, \lambda x\ \sin f(x)\rangle\texttt{)}$

$\Longrightarrow \langle\{f \mapsto \lambda x\ 2x^3, f' \mapsto \lambda x\ 6x^2\},$
$\lambda x\, f'(x)\cos f(x)\rangle$

$\texttt{(map-closure}$
$f\ \langle\{x_1 \mapsto v_1, \ldots\}, e\rangle\texttt{)} \Longrightarrow \langle\{x_1 \mapsto f(v_1), \ldots\}, e\rangle$

need reflective transformation of closure bodies
want transformation done at compile time
need flow analysis

The Essence

$$(\texttt{define}\ (f\ x)\ 2x^3) \rightsquigarrow (\texttt{define}\ (f'\ x)\ 6x^2)$$

$$(\texttt{define}\ (g\ x)\ \sin f(x)) \rightsquigarrow (\texttt{define}\ (g'\ x)\ f'(x)\cos f(x))$$

$$(\mathcal{D}\ g) \Longrightarrow (\mathcal{D}\ \langle\{f \mapsto \lambda x\ 2x^3\}, \lambda x\ \sin f(x)\rangle)$$

$$\Longrightarrow \langle\{f \mapsto \lambda x\ 2x^3, f' \mapsto \lambda x\ 6x^2\}, \lambda x\, f'(x)\cos f(x)\rangle$$

$$(\texttt{map-closure}\ f\ \langle\{x_1 \mapsto v_1, \ldots\}, e\rangle) \Longrightarrow \langle\{x_1 \mapsto f(v_1), \ldots\}, e\rangle$$

need reflective transformation of closure bodies
want transformation done at compile time
need polyvariant flow analysis

Nesting

```
(sqrt (sqrt x))
```

Nesting

```
(sqrt (sqrt x))
```

```
(𝒟 (𝒟 f))
```

Nesting

```
(sqrt (sqrt x))
```

($\mathcal{D}$ ($\mathcal{D}$ f))

```
(map (lambda (x) ... (map (lambda (y) ...) ...) ...) ...)
```

Nesting

```
(sqrt (sqrt x))

(𝒟 (𝒟 f))

(map (lambda (x) ... (map (lambda (y) ...) ...) ...) ...)

(𝒟 (lambda (x) ... (𝒟 (lambda (y) ...) ...) ...) ...)
```

Nesting

```
(sqrt (sqrt x))
```

$(\mathcal{D}$ `(`$\mathcal{D}$ `f))`

```
(map (lambda (x) ... (map (lambda (y) ...) ...) ...) ...)
```

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

$$\max_x \min_y f(x, y)$$

HOME PAGE | TODAY'S PAPER | VIDEO | MOST POPULAR | TIMES TOPICS

My Account | Welcome, qobiyan | Log Out | Help

The New York Times

Business Computing

Search All NYTimes.com
Go

WORLD | U.S. | N.Y. / REGION | BUSINESS | TECHNOLOGY | SCIENCE | HEALTH | SPORTS | OPINION | ARTS | STYLE | TRAVEL | **JOBS** | **REAL ESTATE** | **AUTOS**

Search Technology Go

Inside Technology
Internet | Start-Ups | Business Computing | Companies

Bits Blog »

Personal Tech »
Cellphones, Cameras, Computers and more

Data Analysts Captivated by R's Power

More Articles in Technology »

Stuart Isett for The New York Times

R first appeared in 1996, when the statistics professors Robert Gentleman, left, and Ross Ihaka released the code as a free

The Essence of Forward-Mode AD

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

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

The Essence of Forward-Mode AD

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

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

The Essence of Forward-Mode AD

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

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

- evaluate f

The Essence of Forward-Mode AD

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

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

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

The Essence of Forward-Mode AD

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

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

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

The Essence of Forward-Mode AD

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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).

Arithmetic on Complex Numbers

$a + b\mathrm{i}$

Hamilton, W. R. (1837). *Theory of conjugate functions, or algebraic couples; with a preliminary and elementary essay on algebra as the science of pure time*. Transactions of the Royal Irish Academy, **17**(1):293–422.

Arithmetic on Complex Numbers

$a + b\mathrm{i}$
$\mathrm{i}^2 = -1$

Hamilton, W. R. (1837). *Theory of conjugate functions, or algebraic couples; with a preliminary and elementary essay on algebra as the science of pure time*. Transactions of the Royal Irish Academy, **17**(1):293–422.

Arithmetic on Complex Numbers

$a + b\mathrm{i}$
$\mathrm{i}^2 = -1$

$(a_1 + b_1\mathrm{i}) + (a_2 + b_2\mathrm{i}) = (a_1 + a_2) + (b_1 + b_2)\mathrm{i}$
$(a_1 + b_1\mathrm{i}) \times (a_2 + b_2\mathrm{i}) = (a_1 \times a_2) + (a_1 \times b_2 + a_2 \times b_1)\mathrm{i} + (b_1 \times b_2)\mathrm{i}^2$

Hamilton, W. R. (1837). *Theory of conjugate functions, or algebraic couples; with a preliminary and elementary essay on algebra as the science of pure time*. Transactions of the Royal Irish Academy, **17**(1):293–422.

Arithmetic on Complex Numbers

$a + b\mathrm{i}$
$\mathrm{i}^2 = -1$

$(a_1 + b_1\mathrm{i}) + (a_2 + b_2\mathrm{i}) = (a_1 + a_2) + (b_1 + b_2)\mathrm{i}$
$(a_1 + b_1\mathrm{i}) \times (a_2 + b_2\mathrm{i}) = (a_1 \times a_2) + (a_1 \times b_2 + a_2 \times b_1)\mathrm{i} + (b_1 \times b_2)\mathrm{i}^2$

Hamilton, W. R. (1837). *Theory of conjugate functions, or algebraic couples; with a preliminary and elementary essay on algebra as the science of pure time*. Transactions of the Royal Irish Academy, **17**(1):293–422.

Arithmetic on Complex Numbers

$a + b\mathrm{i}$
$\mathrm{i}^2 = -1$

$(a_1 + b_1\mathrm{i}) + (a_2 + b_2\mathrm{i}) = (a_1 + a_2) + (b_1 + b_2)\mathrm{i}$
$(a_1 + b_1\mathrm{i}) \times (a_2 + b_2\mathrm{i}) = (a_1 \times a_2 - b_1 \times b_2) + (a_1 \times b_2 + a_2 \times b_1)\mathrm{i}$

Hamilton, W. R. (1837). *Theory of conjugate functions, or algebraic couples; with a preliminary and elementary essay on algebra as the science of pure time*. Transactions of the Royal Irish Academy, **17**(1):293–422.

Arithmetic on Complex Numbers

$a + b\mathrm{i}$
$\mathrm{i}^2 = -1$

$(a_1 + b_1\mathrm{i}) + (a_2 + b_2\mathrm{i}) = (a_1 + a_2) + (b_1 + b_2)\mathrm{i}$
$(a_1 + b_1\mathrm{i}) \times (a_2 + b_2\mathrm{i}) = (a_1 \times a_2 - b_1 \times b_2) + (a_1 \times b_2 + a_2 \times b_1)\mathrm{i}$

$\langle a, b \rangle$

Hamilton, W. R. (1837). *Theory of conjugate functions, or algebraic couples; with a preliminary and elementary essay on algebra as the science of pure time*. Transactions of the Royal Irish Academy, **17**(1):293–422.

Arithmetic on Complex Numbers

$a + b\mathrm{i}$
$\mathrm{i}^2 = -1$

$(a_1 + b_1\mathrm{i}) + (a_2 + b_2\mathrm{i}) = (a_1 + a_2) + (b_1 + b_2)\mathrm{i}$
$(a_1 + b_1\mathrm{i}) \times (a_2 + b_2\mathrm{i}) = (a_1 \times a_2 - b_1 \times b_2) + (a_1 \times b_2 + a_2 \times b_1)\mathrm{i}$

$\langle a, b\rangle$

$\langle a_1, b_1\rangle + \langle a_2, b_2\rangle = \langle (a_1 + a_2), (b_1 + b_2)\rangle$
$\langle a_1, b_1\rangle \times \langle a_2, b_2\rangle = \langle (a_1 \times a_2 - b_1 \times b_2), (a_1 \times b_2 + a_2 \times b_1)\rangle$

Hamilton, W. R. (1837). *Theory of conjugate functions, or algebraic couples; with a preliminary and elementary essay on algebra as the science of pure time*. Transactions of the Royal Irish Academy, **17**(1):293–422.

Arithmetic on Dual Numbers

$x + x'\varepsilon$

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

Arithmetic on Dual Numbers

$x + x'\varepsilon$
$\varepsilon^2 = 0$, but $\varepsilon \neq 0$

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

Arithmetic on Dual Numbers

$x + x'\varepsilon$
$\varepsilon^2 = 0$, but $\varepsilon \neq 0$

$(x_1 + x'_1\varepsilon) + (x_2 + x'_2\varepsilon) = (x_1 + x_2) + (x'_1 + x'_2)\varepsilon$
$(x_1 + x'_1\varepsilon) \times (x_2 + x'_2\varepsilon) = (x_1 \times x_2) + (x_1 \times x'_2 + x_2 \times x'_1)\varepsilon + (x'_1 + x'_2)\varepsilon^2$

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

Arithmetic on Dual Numbers

$x + x'\varepsilon$

$\varepsilon^2 = 0$, but $\varepsilon \neq 0$

$$(x_1 + x_1'\varepsilon) + (x_2 + x_2'\varepsilon) = (x_1 + x_2) + (x_1' + x_2')\varepsilon$$
$$(x_1 + x_1'\varepsilon) \times (x_2 + x_2'\varepsilon) = (x_1 \times x_2) + (x_1 \times x_2' + x_2 \times x_1')\varepsilon + (x_1' + x_2')\varepsilon^2$$

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

Arithmetic on Dual Numbers

$x + x'\varepsilon$
$\varepsilon^2 = 0$, but $\varepsilon \neq 0$

$(x_1 + x_1'\varepsilon) + (x_2 + x_2'\varepsilon) = (x_1 + x_2) + (x_1' + x_2')\varepsilon$
$(x_1 + x_1'\varepsilon) \times (x_2 + x_2'\varepsilon) = (x_1 \times x_2) + (x_1 \times x_2' + x_2 \times x_1')\varepsilon$

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

Arithmetic on Dual Numbers

$x + x'\varepsilon$
$\varepsilon^2 = 0$, but $\varepsilon \neq 0$

$(x_1 + x'_1\varepsilon) + (x_2 + x'_2\varepsilon) = (x_1 + x_2) + (x'_1 + x'_2)\varepsilon$
$(x_1 + x'_1\varepsilon) \times (x_2 + x'_2\varepsilon) = (x_1 \times x_2) + (x_1 \times x'_2 + x_2 \times x'_1)\varepsilon$

$\langle x, x' \rangle$

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

Arithmetic on Dual Numbers

$x + x'\varepsilon$
$\varepsilon^2 = 0$, but $\varepsilon \neq 0$

$$(x_1 + x_1'\varepsilon) + (x_2 + x_2'\varepsilon) = (x_1 + x_2) + (x_1' + x_2')\varepsilon$$
$$(x_1 + x_1'\varepsilon) \times (x_2 + x_2'\varepsilon) = (x_1 \times x_2) + (x_1 \times x_2' + x_2 \times x_1')\varepsilon$$

$\langle x, x' \rangle$

$$\langle x_1, x_1' \rangle + \langle x_2, x_2' \rangle = \langle (x_1 + x_2), (x_1' + x_2') \rangle$$
$$\langle x_1, x_1' \rangle \times \langle x_2, x_2' \rangle = \langle (x_1 \times x_2), (x_1 \times x_2' + x_2 \times x_1') \rangle$$

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

Dynamic Overloading: SCMUTILS

```
(define-structure bundle primal tangent)
(define (primal p) (if (bundle? p) (bundle-primal p) p))
(define (tangent p) (if (bundle? p) (bundle-tangent p) 0))

(define +
 (let ((+ +))
  (lambda (x1 x2)
   (make-bundle (+ (primal x1) (primal x2))
                (+ (tangent x1) (tangent x2))))))

(define *
 (let ((+ +) (* *))
  (lambda (x1 x2)
   (make-bundle (* (primal x1) (primal x2))
                (+ (* (primal x1) (tangent x2))
                   (* (tangent x1) (primal x2)))))))

(define ((D f) x) (tangent (f (make-bundle x 1))))
```

Dynamic Overloading: SCMUTILS

```
(define-structure bundle primal tangent)
(define (primal p) (if (bundle? p) (bundle-primal p) p))
(define (tangent p) (if (bundle? p) (bundle-tangent p) 0))

(define +
 (let ((+ +))
  (lambda (x1 x2)
   (make-bundle (+ (primal x1) (primal x2))
                (+ (tangent x1) (tangent x2))))))

(define *
 (let ((+ +) (* *))
  (lambda (x1 x2)
   (make-bundle (* (primal x1) (primal x2))
                (+ (* (primal x1) (tangent x2))
                   (* (tangent x1) (primal x2)))))))

(define ((D f) x) (tangent (f (make-bundle x 1))))

(define (f x) (* 2 (* x (* x x))))
```

Dynamic Overloading: SCMUTILS

```
(define-structure bundle primal tangent)
(define (primal p) (if (bundle? p) (bundle-primal p) p))
(define (tangent p) (if (bundle? p) (bundle-tangent p) 0))

(define +
 (let ((+ +))
  (lambda (x1 x2)
   (make-bundle (+ (primal x1) (primal x2))
                (+ (tangent x1) (tangent x2))))))

(define *
 (let ((+ +) (* *))
  (lambda (x1 x2)
   (make-bundle (* (primal x1) (primal x2))
                (+ (* (primal x1) (tangent x2))
                   (* (tangent x1) (primal x2)))))))

(define ((D f) x) (tangent (f (make-bundle x 1))))

(define (f x) (* 2 (* x (* x x))))

(𝒟 f)
```

Dynamic Overloading: SCMUTILS

```
(define-structure bundle primal tangent)
(define (primal p) (if (bundle? p) (bundle-primal p) p))
(define (tangent p) (if (bundle? p) (bundle-tangent p) 0))

(define +
 (let ((+ +))
  (lambda (x1 x2)
   (make-bundle (+ (primal x1) (primal x2))
                (+ (tangent x1) (tangent x2))))))

(define *
 (let ((+ +) (* *))
  (lambda (x1 x2)
   (make-bundle (* (primal x1) (primal x2))
                (+ (* (primal x1) (tangent x2))
                   (* (tangent x1) (primal x2)))))))

(define ((D f) x) (tangent (f (make-bundle x 1))))

(define (f x) (* 2 (* x (* x x))))

(𝒟 f)
(𝒟 (𝒟 f))
(𝒟 (lambda (x) ... (𝒟 (lambda (y) ...) ...) ...) ...)
```

Dynamic Overloading: SCMUTILS

```
(define-structure bundle primal tangent)
(define (primal p) (if (bundle? p) (bundle-primal p) p))
(define (tangent p) (if (bundle? p) (bundle-tangent p) 0))

(define +
 (let ((+ +))
  (lambda (x1 x2)
   (make-bundle (+ (primal x1) (primal x2))
                (+ (tangent x1) (tangent x2))))))

(define *
 (let ((+ +) (* *))
  (lambda (x1 x2)
   (make-bundle (* (primal x1) (primal x2))
                (+ (* (primal x1) (tangent x2))
                   (* (tangent x1) (primal x2)))))))

(define ((D f) x) (tangent (f (make-bundle x 1))))

(define (f x) (* 2 (* x (* x x))))

(𝒟 f)
(𝒟 (𝒟 f))
(𝒟 (lambda (x) ... (𝒟 (lambda (y) ...) ...) ...) ...)
```

Dynamic Overloading: SCMUTILS

```
(define-structure bundle primal tangent)
(define (primal p) (if (bundle? p) (bundle-primal p) p))
(define (tangent p) (if (bundle? p) (bundle-tangent p) 0))

(define +
 (let ((+ +))
  (lambda (x1 x2)
   (make-bundle (+ (primal x1) (primal x2))
                (+ (tangent x1) (tangent x2))))))

(define *
 (let ((+ +) (* *))
  (lambda (x1 x2)
   (make-bundle (* (primal x1) (primal x2))
                (+ (* (primal x1) (tangent x2))
                   (* (tangent x1) (primal x2)))))))

(define ((D f) x) (tangent (f (make-bundle x 1))))

(define (f x) (* 2 (* x (* x x))))

(𝒟 f)
(𝒟 (𝒟 f))
(𝒟 (lambda (x) ... (𝒟 (lambda (y) ...) ...) ...) ...)
```

Convenient

Dynamic Overloading: SCMUTILS

```
(define-structure bundle primal tangent)
(define (primal p) (if (bundle? p) (bundle-primal p) p))
(define (tangent p) (if (bundle? p) (bundle-tangent p) 0))

(define +
 (let ((+ +))
  (lambda (x1 x2)
   (make-bundle (+ (primal x1) (primal x2))
                (+ (tangent x1) (tangent x2))))))

(define *
 (let ((+ +) (* *))
  (lambda (x1 x2)
   (make-bundle (* (primal x1) (primal x2))
                (+ (* (primal x1) (tangent x2))
                   (* (tangent x1) (primal x2)))))))

(define ((D f) x) (tangent (f (make-bundle x 1))))

(define (f x) (* 2 (* x (* x x))))

(𝒟 f)
(𝒟 (𝒟 f))
(𝒟 (lambda (x) ... (𝒟 (lambda (y) ...) ...) ...) ...)
```

Convenient but slow

Dynamic Overloading: SCMUTILS

```
(define-structure bundle primal tangent)
(define (primal p) (if (bundle? p) (bundle-primal p) p))
(define (tangent p) (if (bundle? p) (bundle-tangent p) 0))

(define +
 (let ((+ +))
  (lambda (x1 x2)
   (make-bundle (+ (primal x1) (primal x2))
                (+ (tangent x1) (tangent x2))))))

(define *
 (let ((+ +) (* *))
  (lambda (x1 x2)
   (make-bundle (* (primal x1) (primal x2))
                (+ (* (primal x1) (tangent x2))
                   (* (tangent x1) (primal x2)))))))

(define ((D f) x) (tangent (f (make-bundle x 1))))

(define (f x) (* 2 (* x (* x x))))

(𝒟 f)
(𝒟 (𝒟 f))
(𝒟 (lambda (x) ... (𝒟 (lambda (y) ...) ...) ...) ...)
```

Convenient but slow

Dynamic Overloading: SCMUTILS

```
(define-structure bundle primal tangent)
(define (primal p) (if (bundle? p) (bundle-primal p) p))
(define (tangent p) (if (bundle? p) (bundle-tangent p) 0))

(define +
 (let ((+ +))
  (lambda (x1 x2)
   (make-bundle (+ (primal x1) (primal x2))
                (+ (tangent x1) (tangent x2))))))

(define *
 (let ((+ +) (* *))
  (lambda (x1 x2)
   (make-bundle (* (primal x1) (primal x2))
                (+ (* (primal x1) (tangent x2))
                   (* (tangent x1) (primal x2)))))))

(define ((D f) x) (tangent (f (make-bundle x 1))))

(define (f x) (* 2 (* x (* x x))))

(𝒟 f)
(𝒟 (𝒟 f))
(𝒟 (lambda (x) ... (𝒟 (lambda (y) ...) ...) ...) ...)
```

Convenient but slow

Dynamic Overloading: SCMUTILS

```
(define-structure bundle primal tangent)
(define (primal p) (if (bundle? p) (bundle-primal p) p))
(define (tangent p) (if (bundle? p) (bundle-tangent p) 0))

(define ((D f) x)
 (fluid-let ((+ (lambda (x1 x2)
                 (make-bundle (+ (primal x1) (primal x2))
                              (+ (tangent x1) (tangent x2)))))
             (* (lambda (x1 x2)
                 (make-bundle (* (primal x1) (primal x2))
                              (+ (* (primal x1) (tangent x2))
                                 (* (tangent x1) (primal x2)))))))
  (tangent (f (make-bundle x 1)))))
```

```
(define (f x) (* 2 (* x (* x x))))

(𝒟 f)
(𝒟 (𝒟 f))
(𝒟 (lambda (x) ... (𝒟 (lambda (y) ...) ...) ...) ...)
```

Convenient but slow

Dynamic Overloading: SCMUTILS

```
(define-structure bundle primal tangent)
(define (primal p) (if (bundle? p) (bundle-primal p) p))
(define (tangent p) (if (bundle? p) (bundle-tangent p) 0))

(define ((D f) x)
 (fluid-let ((+ (lambda (x1 x2)
                 (make-bundle (+ (primal x1) (primal x2))
                              (+ (tangent x1) (tangent x2)))))
             (* (lambda (x1 x2)
                 (make-bundle (* (primal x1) (primal x2))
                              (+ (* (primal x1) (tangent x2))
                                 (* (tangent x1) (primal x2)))))))
  (tangent (f (make-bundle x 1)))))
```

```
(define (f x) (* 2 (* x (* x x))))

(𝒟 f)
(𝒟 (𝒟 f))
(𝒟 (lambda (x) ... (𝒟 (lambda (y) ...) ...) ...) ...)
```

Convenient but slow

Preprocessor: ADIFOR and TAPENADE

```
function f(x)
double precision x, f
f = 2.0d0*x*x*x
end
```

Preprocessor: ADIFOR and TAPENADE

```
function f(x)
double precision x, f
f = 2.0d0*x*x*x
end

function gf(x, gx, gresult)
double precision x, gx, gf, gresult
gf = 2.0d0*x*x*x
gresult = 6.0d0*x*x*gx
end
```

Preprocessor: ADIFOR and TAPENADE

```
function f(x)
double precision x, f
f = 2.0d0*x*x*x
end

function gf(x, gx, gresult)
double precision x, gx, gf, gresult
gf = 2.0d0*x*x*x
gresult = 6.0d0*x*x*gx
end
```

Fast

Preprocessor: ADIFOR and TAPENADE

```
function f(x)
double precision x, f
f = 2.0d0*x*x*x
end

function gf(x, gx, gresult)
double precision x, gx, gf, gresult
gf = 2.0d0*x*x*x
gresult = 6.0d0*x*x*gx
end
```

Fast but inconvenient

Preprocessor: ADIFOR and TAPENADE

```
function f(x)
double precision x, f
f = 2.0d0*x*x*x
end
```

```
AD_TOP = f
```

```
function gf(x, gx, gresult)
double precision x, gx, gf, gresult
gf = 2.0d0*x*x*x
gresult = 6.0d0*x*x*gx
end
```

Fast but inconvenient

Preprocessor: ADIFOR and TAPENADE

```
function f(x)
double precision x, f
f = 2.0d0*x*x*x
end

function gf(x, gx, gresult)
double precision x, gx, gf, gresult
gf = 2.0d0*x*x*x
gresult = 6.0d0*x*x*gx
end
```

```
AD_TOP = f
AD_IVARS = x
AD_DVARS = f
```

Fast but inconvenient

Preprocessor: ADIFOR and TAPENADE

```
function f(x)
double precision x, f
f = 2.0d0*x*x*x
end
```

```
AD_TOP = f
AD_IVARS = x
AD_DVARS = f
```

```
function gf(x, gx, gresult)
double precision x, gx, gf, gresult
gf = 2.0d0*x*x*x
gresult = 6.0d0*x*x*gx
end
```

Fast but inconvenient

Preprocessor: ADIFOR and TAPENADE

```
function f(x)
double precision x, f
f = 2.0d0*x*x*x
end
```

```
AD_TOP = f
AD_IVARS = x
AD_DVARS = f
```

```
function gf(x, gx, gresult)
double precision x, gx, gf, gresult
gf = 2.0d0*x*x*x
gresult = 6.0d0*x*x*gx
end
```

Fast but inconvenient

Preprocessor: ADIFOR and TAPENADE

```
function f(x)
double precision x, f
f = 2.0d0*x*x*x
end
```

```
AD_TOP = f
AD_IVARS = x
AD_DVARS = f
```

```
function gf(x, gx, gresult)
double precision x, gx, gf, gresult
gf = 2.0d0*x*x*x
gresult = 6.0d0*x*x*gx
end
```

```
AD_TOP = gf
AD_IVARS = x, gx
AD_DVARS = gf, gresult
```

Fast but inconvenient

Preprocessor: ADIFOR and TAPENADE

```
function f(x)
double precision x, f
f = 2.0d0*x*x*x
end
```

```
AD_TOP = f
AD_IVARS = x
AD_DVARS = f
```

```
function gf(x, gx, gresult)
double precision x, gx, gf, gresult
gf = 2.0d0*x*x*x
gresult = 6.0d0*x*x*gx
end
```

```
AD_TOP = gf
AD_IVARS = x, gx
AD_DVARS = gf, gresult
```

```
function ggf(x, gx, gx, ggx, gresult, ggresult, gresult)
double precision x, gx, gx, ggx, ggf, gresult, gresult, ggresult
ggf = 2.0d0*x*x*x
gresult = 6.0d0*x*x*gx
gresult = 6.0d0*x*x*gx
ggresult = 6.0d0*x*x*ggx+12.0d0*x*gx*gx
end
```

Fast but inconvenient

Preprocessor: ADIFOR and TAPENADE

```
function f(x)
double precision x, f
f = 2.0d0*x*x*x
end
```

```
AD_TOP = f
AD_IVARS = x
AD_DVARS = f
```

```
function gf(x, gx, gresult)
double precision x, gx, gf, gresult
gf = 2.0d0*x*x*x
gresult = 6.0d0*x*x*gx
end
```

```
AD_TOP = gf
AD_IVARS = x, gx
AD_DVARS = gf, gresult
```

```
function ggf(x, gx, gx, ggx, gresult, ggresult, gresult)
double precision x, gx, gx, ggx, ggf, gresult, gresult, ggresult
ggf = 2.0d0*x*x*x
gresult = 6.0d0*x*x*gx
gresult = 6.0d0*x*x*gx
ggresult = 6.0d0*x*x*ggx+12.0d0*x*gx*gx
end
```

Fast but inconvenient

Preprocessor: ADIFOR and TAPENADE

```
function f(x)
double precision x, f
f = 2.0d0*x*x*x
end
```

```
AD_TOP = f
AD_IVARS = x
AD_DVARS = f
```

```
function gf(x, gx, gresult)
double precision x, gx, gf, gresult
gf = 2.0d0*x*x*x
gresult = 6.0d0*x*x*gx
end
```

```
AD_TOP = gf
AD_IVARS = x, gx
AD_DVARS = gf, gresult
```

```
function ggf(x, gx, gx, ggx, gresult, ggresult, gresult)
double precision x, gx, gx, ggx, ggf, gresult, gresult, ggresult
ggf = 2.0d0*x*x*x
gresult = 6.0d0*x*x*gx
gresult = 6.0d0*x*x*gx
ggresult = 6.0d0*x*x*ggx+12.0d0*x*gx*gx
end
```

Fast but inconvenient

Preprocessor: ADIFOR and TAPENADE

```
function f(x)                               AD_TOP = f
double precision x, f                       AD_IVARS = x
f = 2.0d0*x*x*x                             AD_DVARS = f
end

function gf(x, gx, gresult)                 AD_TOP = gf
double precision x, gx, gf, gresult         AD_IVARS = x, gx
gf = 2.0d0*x*x*x                            AD_DVARS = gf, gresult
gresult = 6.0d0*x*x*gx
end

function ggf(x, gx, gx, ggx, gresult, ggresult, gresult)
double precision x, gx, gx, ggx, ggf, gresult, gresult, ggresult
ggf = 2.0d0*x*x*x
gresult = 6.0d0*x*x*gx
gresult = 6.0d0*x*x*gx
ggresult = 6.0d0*x*x*ggx+12.0d0*x*gx*gx
end
```

Fast but inconvenient

Preprocessor: ADIFOR and TAPENADE

```
function f(x)
double precision x, f
f = 2.0d0*x*x*x
end
```

```
AD_TOP = f
AD_IVARS = x
AD_DVARS = f
```

```
function gf(x, gx, gresult)
double precision x, gx, gf, gresult
gf = 2.0d0*x*x*x
gresult = 6.0d0*x*x*gx
end
```

```
AD_TOP = gf
AD_IVARS = x, gx
AD_DVARS = gf, gresult
AD_PREFIX = h
```

```
function hgf(x, hx, gx, hgx, gresult, hgresult, hresult)
double precision x, hx, gx, hgx, hgf, hresult, gresult, hgresult
hgf = 2.0d0*x*x*x
hresult = 6.0d0*x*x*hx
gresult = 6.0d0*x*x*gx
hgresult = 6.0d0*x*x*hgx+12.0d0*x*gx*hx
end
```

Fast but inconvenient

Static Overloading: FADBAD++

```
double f(double x) {return 2*x*x*x;}
double x;
... f(x) ...
```

Static Overloading: FADBAD++

```
double f(double x) {return 2*x*x*x;}
double x;
... f(x) ...

F<double> f(F<double> x) {return 2*x*x*x;}
F<double> x;
x.diff(0, 1);
... f(x).d(0) ...
```

Static Overloading: FADBAD++

```
double f(double x) {return 2*x*x*x;}
double x;
... f(x) ...

F<double> f(F<double> x) {return 2*x*x*x;}
F<double> x;
x.diff(0, 1);
... f(x).d(0) ...

F<F<double> > f(F<F<double> > x) {return 2*x*x*x;}
F<F<double> > x;
x.diff(0, 1);
x.diff(0, 1).diff(0,1);
... f(x).d(0).d(0) ...
```

Static Overloading: FADBAD++

```
double f(double x) {return 2*x*x*x;}
double x;
... f(x) ...

F<double> f(F<double> x) {return 2*x*x*x;}
F<double> x;
x.diff(0, 1);
... f(x).d(0) ...

F<F<double> > f(F<F<double> > x) {return 2*x*x*x;}
F<F<double> > x;
x.diff(0, 1);
x.diff(0, 1).diff(0,1);
... f(x).d(0).d(0) ...
```

Slow

Static Overloading: FADBAD++

```
double f(double x) {return 2*x*x*x;}
double x;
... f(x) ...

F<double> f(F<double> x) {return 2*x*x*x;}
F<double> x;
x.diff(0, 1);
... f(x).d(0) ...

F<F<double> > f(F<F<double> > x) {return 2*x*x*x;}
F<F<double> > x;
x.diff(0, 1);
x.diff(0, 1).diff(0,1);
... f(x).d(0).d(0) ...
```

Slow

Static Overloading: FADBAD++

```
double f(double x) {return 2*x*x*x;}
double x;
... f(x) ...

F<double> f(F<double> x) {return 2*x*x*x;}
F<double> x;
x.diff(0, 1);
... f(x).d(0) ...

F<F<double> > f(F<F<double> > x) {return 2*x*x*x;}
F<F<double> > x;
x.diff(0, 1);
x.diff(0, 1).diff(0,1);
... f(x).d(0).d(0) ...
```

Slow

Static Overloading: FADBAD++

```
double f(double x) {return 2*x*x*x;}
double x;
... f(x) ...

F<double> f(F<double> x) {return 2*x*x*x;}
F<double> x;
x.diff(0, 1);
... f(x).d(0) ...

F<F<double> > f(F<F<double> > x) {return 2*x*x*x;}
F<F<double> > x;
x.diff(0, 1);
x.diff(0, 1).diff(0,1);
... f(x).d(0).d(0) ...
```

Slow

Static Overloading: FADBAD++

```
double f(double x) {return 2*x*x*x;}
double x;
... f(x) ...

F<double> f(F<double> x) {return 2*x*x*x;}
F<double> x;
x.diff(0, 1);
... f(x).d(0) ...

F<F<double> > f(F<F<double> > x) {return 2*x*x*x;}
F<F<double> > x;
x.diff(0, 1);
x.diff(0, 1).diff(0,1);
... f(x).d(0).d(0) ...
```

Slow and inconvenient

Static Overloading: FADBAD++

```
double f(double x) {return 2*x*x*x;}
double x;
... f(x) ...

F<double> f(F<double> x) {return 2*x*x*x;}
F<double> x;
x.diff(0, 1);
... f(x).d(0) ...

F<F<double> > f(F<F<double> > x) {return 2*x*x*x;}
F<F<double> > x;
x.diff(0, 1);
x.diff(0, 1).diff(0,1);
... f(x).d(0).d(0) ...
```

Slow and inconvenient

Static Overloading: FADBAD++

```
double f(double x) {return 2*x*x*x;}
double x;
... f(x) ...

F<double> f(F<double> x) {return 2*x*x*x;}
F<double> x;
x.diff(0, 1);
... f(x).d(0) ...

F<F<double> > f(F<F<double> > x) {return 2*x*x*x;}
F<F<double> > x;
x.diff(0, 1);
x.diff(0, 1).diff(0,1);
... f(x).d(0).d(0) ...
```

Slow and inconvenient

Static Overloading: FADBAD++

```
double f(double x) {return 2*x*x*x;}
double x;
... f(x) ...

F<double> f(F<double> x) {return 2*x*x*x;}
F<double> x;
x.diff(0, 1);
... f(x).d(0) ...

F<F<double> > f(F<F<double> > x) {return 2*x*x*x;}
F<F<double> > x;
x.diff(0, 1);
x.diff(0, 1).diff(0,1);
... f(x).d(0).d(0) ...
```

Slow and inconvenient

Static Overloading: FADBAD++

```
double f(double x) {return 2*x*x*x;}
double x;
... f(x) ...

F<double> f(F<double> x) {return 2*x*x*x;}
F<double> x;
x.diff(0, 1);
... f(x).d(0) ...

F<F<double> > f(F<F<double> > x) {return 2*x*x*x;}
F<F<double> > x;
x.diff(0, 1);
x.diff(0, 1).diff(0,1);
... f(x).d(0).d(0) ...
```

Slow and inconvenient

Static Overloading: FADBAD++

```
double f(double x) {return 2*x*x*x;}
double x;
... f(x) ...

F<double> f(F<double> x) {return 2*x*x*x;}
F<double> x;
x.diff(0, 1);
... f(x).d(0) ...

F<F<double> > f(F<F<double> > x) {return 2*x*x*x;}
F<F<double> > x;
x.diff(0, 1);
x.diff(0, 1).diff(0,1);
... f(x).d(0).d(0) ...
```

Slow and inconvenient

Static Overloading: FADBAD++

```
double f(double x) {return 2*x*x*x;}
double x;
... f(x) ...

F<double> f(F<double> x) {return 2*x*x*x;}
F<double> x;
x.diff(0, 1);
... f(x).d(0) ...

F<F<double> > f(F<F<double> > x) {return 2*x*x*x;}
F<F<double> > x;
x.diff(0, 1);
x.diff(0, 1).diff(0,1);
... f(x).d(0).d(0) ...
```

Slow and inconvenient

Static Overloading: FADBAD++

```
double f(double x) {return 2*x*x*x;}
double x;
... f(x) ...

F<double> f(F<double> x) {return 2*x*x*x;}
F<double> x;
x.diff(0, 1);
... f(x).d(0) ...

F<F<double> > f(F<F<double> > x) {return 2*x*x*x;}
F<F<double> > x;
x.diff(0, 1);
x.diff(0, 1).diff(0,1);
... f(x).d(0).d(0) ...

template <typename T>
T f(T x) {return 2*x*x*x;}
T x;
```

Slow and inconvenient

Static Overloading: FADBAD++

```
double f(double x) {return 2*x*x*x;}
double x;
... f(x) ...

F<double> f(F<double> x) {return 2*x*x*x;}
F<double> x;
x.diff(0, 1);
... f(x).d(0) ...

F<F<double> > f(F<F<double> > x) {return 2*x*x*x;}
F<F<double> > x;
x.diff(0, 1);
x.diff(0, 1).diff(0,1);
... f(x).d(0).d(0) ...

template <typename T>
T f(T x) {return 2*x*x*x;}
T x;
```

Slow and inconvenient

Static Overloading: FADBAD++

```
double f(double x) {return 2*x*x*x;}
double x;
... f(x) ...

F<double> f(F<double> x) {return 2*x*x*x;}
F<double> x;
x.diff(0, 1);
... f(x).d(0) ...

F<F<double> > f(F<F<double> > x) {return 2*x*x*x;}
F<F<double> > x;
x.diff(0, 1);
x.diff(0, 1).diff(0,1);
... f(x).d(0).d(0) ...

template <typename T>
T f(T x) {return 2*x*x*x;}
T x;
```

Slow and inconvenient

Our API for Functional Forward AD

Differential geometry bundles points $\mathbb{R}^n$ in a manifold with tangent vectors $\overrightarrow{\mathbb{R}^n}$

Our API for Functional Forward AD

$$\texttt{bundle} \quad : \quad \mathbb{R}^n \times \overrightarrow{\mathbb{R}^n} \rightarrow (\mathbb{R}^n \triangleright \overrightarrow{\mathbb{R}^n})$$

Differential geometry bundles points $\mathbb{R}^n$ in a manifold with tangent vectors $\overrightarrow{\mathbb{R}^n}$

Our API for Functional Forward AD

$$\texttt{bundle} : \mathbb{R}^n \times \overline{\mathbb{R}^n} \to (\mathbb{R}^n \triangleright \overline{\mathbb{R}^n})$$

$$\texttt{primal} : (\mathbb{R}^n \triangleright \overline{\mathbb{R}^n}) \to \mathbb{R}^n$$

$$\texttt{tangent} : (\mathbb{R}^n \triangleright \overline{\mathbb{R}^n}) \to \overline{\mathbb{R}^n}$$

Differential geometry bundles points $\mathbb{R}^n$ in a manifold with tangent vectors $\overline{\mathbb{R}^n}$

Our API for Functional Forward AD

$$\texttt{bundle} : \mathbb{R}^n \times \overline{\mathbb{R}^{\acute{n}}} \rightarrow (\mathbb{R}^n \triangleright \overline{\mathbb{R}^{\acute{n}}})$$

$$\texttt{primal} : (\mathbb{R}^n \triangleright \overline{\mathbb{R}^{\acute{n}}}) \rightarrow \mathbb{R}^n$$

$$\texttt{tangent} : (\mathbb{R}^n \triangleright \overline{\mathbb{R}^{\acute{n}}}) \rightarrow \overline{\mathbb{R}^{\acute{n}}}$$

$$\texttt{j}\star : (\mathbb{R}^n \rightarrow \mathbb{R}^m) \rightarrow ((\mathbb{R}^n \triangleright \overline{\mathbb{R}^{\acute{n}}}) \rightarrow (\mathbb{R}^m \triangleright \overline{\mathbb{R}^{\acute{m}}}))$$

Differential geometry bundles points $\mathbb{R}^n$ in a manifold with tangent vectors $\overline{\mathbb{R}^{\acute{n}}}$
`j*` maps a function to its *push forward*

Our API for Functional Forward AD

$$
\begin{aligned}
\texttt{bundle} &: \mathbb{R}^n \times \overrightarrow{\mathbb{R}^n} \rightarrow (\mathbb{R}^n \triangleright \overrightarrow{\mathbb{R}^n}) \\
\texttt{primal} &: (\mathbb{R}^n \triangleright \overrightarrow{\mathbb{R}^n}) \rightarrow \mathbb{R}^n \\
\texttt{tangent} &: (\mathbb{R}^n \triangleright \overrightarrow{\mathbb{R}^n}) \rightarrow \overrightarrow{\mathbb{R}^n} \\
\texttt{j*} &: (\mathbb{R}^n \rightarrow \mathbb{R}^m) \rightarrow ((\mathbb{R}^n \triangleright \overrightarrow{\mathbb{R}^n}) \rightarrow (\mathbb{R}^m \triangleright \overrightarrow{\mathbb{R}^m}))
\end{aligned}
$$

Differential geometry bundles points $\mathbb{R}^n$ in a manifold with tangent vectors $\overrightarrow{\mathbb{R}^n}$

`j*` maps a function to its *push forward*

Our API for Functional Forward AD

$$\texttt{bundle} : \mathbb{R}^n \times \overrightarrow{\mathbb{R}^n} \to (\mathbb{R}^n \triangleright \overrightarrow{\mathbb{R}^n})$$
$$\texttt{primal} : (\mathbb{R}^n \triangleright \overrightarrow{\mathbb{R}^n}) \to \mathbb{R}^n$$
$$\texttt{tangent} : (\mathbb{R}^n \triangleright \overrightarrow{\mathbb{R}^n}) \to \overrightarrow{\mathbb{R}^n}$$
$$\texttt{j}\star : (\mathbb{R}^n \to \mathbb{R}^m) \to ((\mathbb{R}^n \triangleright \overrightarrow{\mathbb{R}^n}) \to (\mathbb{R}^m \triangleright \overrightarrow{\mathbb{R}^m}))$$

Differential geometry bundles points $\mathbb{R}^n$ in a manifold with tangent vectors $\overrightarrow{\mathbb{R}^n}$
`j*` maps a function to its *push forward*

Our API for Functional Forward AD

$$\texttt{bundle} : \tau \times \overrightarrow{\tau} \to (\tau \triangleright \overrightarrow{\tau})$$

$$\texttt{primal} : (\tau \triangleright \overrightarrow{\tau}) \to \tau$$

$$\texttt{tangent} : (\tau \triangleright \overrightarrow{\tau}) \to \overrightarrow{\tau}$$

$$\texttt{j}\star : (\tau_1 \to \tau_2) \to ((\tau_1 \triangleright \overrightarrow{\tau_1}) \to (\tau_2 \triangleright \overrightarrow{\tau_2}))$$

Differential geometry bundles points $\mathbb{R}^n$ in a manifold with tangent vectors $\overrightarrow{\mathbb{R}^n}$
`j⋆` maps a function to its *push forward*
Generalize to arbitrary types

Our API for Functional Forward AD

$$
\begin{aligned}
\texttt{bundle} &: \tau \times \overline{\tau'} \rightarrow (\tau \rhd \overline{\tau'})\\
\texttt{primal} &: (\tau \rhd \overline{\tau'}) \rightarrow \tau\\
\texttt{tangent} &: (\tau \rhd \overline{\tau'}) \rightarrow \overline{\tau'}\\
\texttt{j*} &: (\tau_1 \rightarrow \tau_2) \rightarrow ((\tau_1 \rhd \overline{\tau_1'}) \rightarrow (\tau_2 \rhd \overline{\tau_2'}))
\end{aligned}
$$

Differential geometry bundles points $\mathbb{R}^n$ in a manifold with tangent vectors $\overline{\mathbb{R}^{n'}}$

`j*` maps a function to its *push forward*

Generalize to arbitrary types

What is the tangent of a discrete value or a function?

Our API for Functional Forward AD

$$
\begin{aligned}
\texttt{bundle} &: \tau \times \overline{\tau'} \rightarrow (\tau \triangleright \overline{\tau'})\\
\texttt{primal} &: (\tau \triangleright \overline{\tau'}) \rightarrow \tau\\
\texttt{tangent} &: (\tau \triangleright \overline{\tau'}) \rightarrow \overline{\tau'}\\
\texttt{j*} &: (\tau_1 \rightarrow \tau_2) \rightarrow ((\tau_1 \triangleright \overline{\tau_1'}) \rightarrow (\tau_2 \triangleright \overline{\tau_2'}))
\end{aligned}
$$

Differential geometry bundles points $\mathbb{R}^n$ in a manifold with tangent vectors $\overline{\mathbb{R}^{n'}}$
`j*` maps a function to its *push forward*
Generalize to arbitrary types
What is the tangent of a discrete value or a function?
Can abbreviate $\tau \triangleright \overline{\tau'}$ as $\overrightarrow{\tau}$

Our API for Functional Forward AD

$$\texttt{bundle} : \tau \times \overline{\tau} \rightarrow \overrightarrow{\tau}$$
$$\texttt{primal} : \overrightarrow{\tau} \rightarrow \tau$$
$$\texttt{tangent} : \overrightarrow{\tau} \rightarrow \overline{\tau}$$
$$\texttt{j*} : (\tau_1 \rightarrow \tau_2) \rightarrow (\overrightarrow{\tau_1} \rightarrow \overrightarrow{\tau_2})$$

Differential geometry bundles points $\mathbb{R}^n$ in a manifold with tangent vectors $\overline{\mathbb{R}^n}$
`j*` maps a function to its *push forward*
Generalize to arbitrary types
What is the tangent of a discrete value or a function?
Can abbreviate $\tau \triangleright \overline{\tau}$ as $\overrightarrow{\tau}$

Our API for Functional Forward AD

$$\begin{aligned}
\texttt{bundle} &: \tau \times \overline{\tau'} \rightarrow \overrightarrow{\tau}\\
\texttt{primal} &: \overrightarrow{\tau} \rightarrow \tau\\
\texttt{tangent} &: \overrightarrow{\tau} \rightarrow \overline{\tau'}\\
\overrightarrow{\mathcal{J}} &: (\tau_1 \rightarrow \tau_2) \rightarrow (\overrightarrow{\tau_1} \rightarrow \overrightarrow{\tau_2})
\end{aligned}$$

Differential geometry bundles points $\mathbb{R}^n$ in a manifold with tangent vectors $\overline{\mathbb{R}^{n'}}$

`j*` maps a function to its *push forward*

Generalize to arbitrary types

What is the tangent of a discrete value or a function?

Can abbreviate $\tau \triangleright \overline{\tau'}$ as $\overrightarrow{\tau}$

Sometimes write `j*` as $\overrightarrow{\mathcal{J}}$

Our API for Functional Forward AD

$$\begin{aligned}
\texttt{bundle} &: \tau \times \overline{\tau} \rightarrow \overrightarrow{\tau} \\
\texttt{primal} &: \overrightarrow{\tau} \rightarrow \tau \\
\texttt{tangent} &: \overrightarrow{\tau} \rightarrow \overline{\tau} \\
\texttt{j}\star &: (\tau_1 \rightarrow \tau_2) \rightarrow (\overrightarrow{\tau_1} \rightarrow \overrightarrow{\tau_2})
\end{aligned}$$

```
(define ((𝒟 f) x) (tangent ((j* f) (bundle x 1))))
```

Differential geometry bundles points $\mathbb{R}^n$ in a manifold with tangent vectors $\overline{\mathbb{R}^n}$
`j*` maps a function to its *push forward*
Generalize to arbitrary types
What is the tangent of a discrete value or a function?
Can abbreviate $\tau \rhd \overline{\tau}$ as $\overrightarrow{\tau}$
Sometimes write `j*` as $\overrightarrow{\mathcal{J}}$

Convenient

Our API for Functional Forward AD

$$
\begin{aligned}
\texttt{bundle} &: \tau \times \overline{\tau} \rightarrow \overrightarrow{\tau} \\
\texttt{primal} &: \overrightarrow{\tau} \rightarrow \tau \\
\texttt{tangent} &: \overrightarrow{\tau} \rightarrow \overline{\tau} \\
\texttt{j*} &: (\tau_1 \rightarrow \tau_2) \rightarrow (\overrightarrow{\tau_1} \rightarrow \overrightarrow{\tau_2})
\end{aligned}
$$

```
(define ((𝒟 f) x) (tangent ((j* f) (bundle x 1))))
(𝒟 f)
```

Differential geometry bundles points $\mathbb{R}^n$ in a manifold with tangent vectors $\overline{\mathbb{R}^n}$
`j*` maps a function to its *push forward*
Generalize to arbitrary types
What is the tangent of a discrete value or a function?
Can abbreviate $\tau \triangleright \overline{\tau}$ as $\overrightarrow{\tau}$
Sometimes write `j*` as $\overrightarrow{\mathcal{J}}$

Convenient

Our API for Functional Forward AD

$$\begin{aligned}
\texttt{bundle} &: \tau \times \overrightarrow{\tau}' \rightarrow \overrightarrow{\tau} \\
\texttt{primal} &: \overrightarrow{\tau} \rightarrow \tau \\
\texttt{tangent} &: \overrightarrow{\tau} \rightarrow \overrightarrow{\tau}' \\
\texttt{j*} &: (\tau_1 \rightarrow \tau_2) \rightarrow (\overrightarrow{\tau_1} \rightarrow \overrightarrow{\tau_2})
\end{aligned}$$

```
(define ((𝒟 f) x) (tangent ((j* f) (bundle x 1))))
(𝒟 f)
(𝒟 (𝒟 f))
```

Differential geometry bundles points $\mathbb{R}^n$ in a manifold with tangent vectors $\overrightarrow{\mathbb{R}^n}'$
`j*` maps a function to its *push forward*
Generalize to arbitrary types
What is the tangent of a discrete value or a function?
Can abbreviate $\tau \triangleright \overrightarrow{\tau}'$ as $\overrightarrow{\tau}$
Sometimes write `j*` as $\overrightarrow{\mathcal{J}}$

Convenient

Our API for Functional Forward AD

$$\begin{aligned}
\texttt{bundle} &: \tau \times \overline{\tau} \rightarrow \overrightarrow{\tau} \\
\texttt{primal} &: \overrightarrow{\tau} \rightarrow \tau \\
\texttt{tangent} &: \overrightarrow{\tau} \rightarrow \overline{\tau} \\
\texttt{j}\star &: (\tau_1 \rightarrow \tau_2) \rightarrow (\overrightarrow{\tau_1} \rightarrow \overrightarrow{\tau_2})
\end{aligned}$$

```
(define ((𝒟 f) x) (tangent ((j* f) (bundle x 1))))
(𝒟 f)
(𝒟 (𝒟 f))
(𝒟 (lambda (x) ... (𝒟 (lambda (y) ...) ...) ...)
```

Differential geometry bundles points $\mathbb{R}^n$ in a manifold with tangent vectors $\overline{\mathbb{R}^n}$
j⋆ maps a function to its *push forward*
Generalize to arbitrary types
What is the tangent of a discrete value or a function?
Can abbreviate $\tau \triangleright \overline{\tau}$ as $\overrightarrow{\tau}$
Sometimes write j⋆ as $\overrightarrow{\mathcal{J}}$

Convenient

Our API for Functional Forward AD

$$\begin{aligned}
\texttt{bundle} &: \tau \times \overline{\tau}' \to \overrightarrow{\tau}\\
\texttt{primal} &: \overrightarrow{\tau} \to \tau\\
\texttt{tangent} &: \overrightarrow{\tau} \to \overline{\tau}'\\
\texttt{j*} &: (\tau_1 \to \tau_2) \to (\overrightarrow{\tau_1} \to \overrightarrow{\tau_2})
\end{aligned}$$

```
(define ((𝒟 f) x) (tangent ((j* f) (bundle x 1))))
(𝒟 f)
(𝒟 (𝒟 f))
(𝒟 (lambda (x) ... (𝒟 (lambda (y) ...) ...) ...) ...)
```

Differential geometry bundles points $\mathbb{R}^n$ in a manifold with tangent vectors $\overline{\mathbb{R}^n}'$
`j*` maps a function to its *push forward*
Generalize to arbitrary types
What is the tangent of a discrete value or a function?
Can abbreviate $\tau \triangleright \overline{\tau}'$ as $\overrightarrow{\tau}$
Sometimes write `j*` as $\overrightarrow{\mathcal{J}}$
What is `(j* j*)`?
Convenient

Our API for Functional Forward AD

$$\begin{aligned} \texttt{bundle} &: \tau \times \overline{\tau} \to \overrightarrow{\tau} \\ \texttt{primal} &: \overrightarrow{\tau} \to \tau \\ \texttt{tangent} &: \overrightarrow{\tau} \to \overline{\tau} \\ \texttt{j*} &: (\tau_1 \to \tau_2) \to (\overrightarrow{\tau_1} \to \overrightarrow{\tau_2}) \end{aligned}$$

```
(define ((𝒟 f) x) (tangent ((j* f) (bundle x 1))))
(𝒟 f)
(𝒟 (𝒟 f))
(𝒟 (lambda (x) ... (𝒟 (lambda (y) ...) ...) ...) ...)
```

Differential geometry bundles points $\mathbb{R}^n$ in a manifold with tangent vectors $\overline{\mathbb{R}^n}$

`j*` maps a function to its *push forward*

Generalize to arbitrary types

What is the tangent of a discrete value or a function?

Can abbreviate $\tau \triangleright \overline{\tau}$ as $\overrightarrow{\tau}$

Sometimes write `j*` as $\overrightarrow{\mathcal{J}}$

What is `(j* j*)`?

Convenient

Our API for Functional Forward AD

$$\begin{aligned}
\texttt{bundle} &: \tau \times \overline{\tau}' \rightarrow \overrightarrow{\tau}\\
\texttt{primal} &: \overrightarrow{\tau} \rightarrow \tau\\
\texttt{tangent} &: \overrightarrow{\tau} \rightarrow \overline{\tau}'\\
\texttt{j*} &: (\tau_1 \rightarrow \tau_2) \rightarrow (\overrightarrow{\tau_1} \rightarrow \overrightarrow{\tau_2})
\end{aligned}$$

```
(define ((𝒟 f) x) (tangent ((j* f) (bundle x 1))))
(𝒟 f)
(𝒟 (𝒟 f))
(𝒟 (lambda (x) ... (𝒟 (lambda (y) ...) ...) ...) ...)
```

Differential geometry bundles points $\mathbb{R}^n$ in a manifold with tangent vectors $\overline{\mathbb{R}^n}'$
`j*` maps a function to its *push forward*
Generalize to arbitrary types
What is the tangent of a discrete value or a function?
Can abbreviate $\tau \triangleright \overline{\tau}'$ as $\overrightarrow{\tau}$
Sometimes write `j*` as $\overrightarrow{\mathcal{J}}$
What is `(j* j*)`?
Convenient and fast

A property

A property

$x : \mathbb{R}^n$

A property

$x : \mathbb{R}^n$ $\overline{x'} : \mathbb{R}^n$

A property

$x : \mathbb{R}^n$ $\qquad \overleftarrow{x'} : \mathbb{R}^n$ $\qquad f : \mathbb{R}^n \rightarrow \mathbb{R}^m$

A property

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

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

A property

$x : \mathbb{R}^n$ $\qquad \overline{x'} : \mathbb{R}^n$ $\qquad f : \mathbb{R}^n \rightarrow \mathbb{R}^m$

$((\mathcal{J}\ f)\ x)[i,j] = \frac{\partial f(x)[i]}{\partial x[j]}$ $\qquad ((\mathcal{J}\ f)\ x) : \mathbb{R}^{m \times n}$

A property

$x : \mathbb{R}^n$ $\quad$ $\overleftarrow{x'} : \mathbb{R}^n$ $\quad$ $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$

$((\mathcal{J}\ f)\ x)[i,j] = \frac{\partial f(x)[i]}{\partial x[j]}$ $\quad$ $((\mathcal{J}\ f)\ x) : \mathbb{R}^{m \times n}$ $\quad$ $((\mathcal{J}\ f)\ x) : \mathbb{R}^n \xrightarrow{L} \mathbb{R}^m$

A property

$x : \mathbb{R}^n$ $\quad\quad \overrightarrow{x'} : \mathbb{R}^n$ $\quad\quad f : \mathbb{R}^n \rightarrow \mathbb{R}^m$

$((\mathcal{J}\ f)\ x)[i,j] = \frac{\partial f(x)[i]}{\partial x[j]}$ $\quad\quad ((\mathcal{J}\ f)\ x) : \mathbb{R}^{m \times n}$ $\quad\quad ((\mathcal{J}\ f)\ x) : \mathbb{R}^n \xrightarrow{L} \mathbb{R}^m$

(primal ((j⋆ f) (bundle x $\overrightarrow{x'}$))) $=$ (f x)

A property

$x : \mathbb{R}^n$ $\qquad \overrightarrow{x'} : \mathbb{R}^n$ $\qquad f : \mathbb{R}^n \to \mathbb{R}^m$

$((\mathcal{J}\ f)\ x)[i,j] = \frac{\partial f(x)[i]}{\partial x[j]}$ $\qquad ((\mathcal{J}\ f)\ x) : \mathbb{R}^{m \times n}$ $\qquad ((\mathcal{J}\ f)\ x) : \mathbb{R}^n \xrightarrow{L} \mathbb{R}^m$

(primal ((j* f) (bundle x $\overrightarrow{x'}$))) $= (f\ \ x)$

(tangent ((j* f) (bundle x $\overrightarrow{x'}$))) $= ((\mathcal{J}\ f)\ x) \times \overrightarrow{x'}$

A property

$x : \mathbb{R}^n$ $\quad \overrightarrow{x'} : \mathbb{R}^n$ $\quad f : \mathbb{R}^n \rightarrow \mathbb{R}^m$

$((\mathcal{J}\ f)\ x)[i,j] = \frac{\partial f(x)[i]}{\partial x[j]}$ $\quad ((\mathcal{J}\ f)\ x) : \mathbb{R}^{m \times n}$ $\quad ((\mathcal{J}\ f)\ x) : \mathbb{R}^n \xrightarrow{L} \mathbb{R}^m$

$(\texttt{primal}\ ((\texttt{j*}\ f)\ (\texttt{bundle}\ x\ \overrightarrow{x'}))) = (f\ x)$

$(\texttt{tangent}\ ((\texttt{j*}\ f)\ (\texttt{bundle}\ x\ \overrightarrow{x'}))) = ((\mathcal{J}\ f)\ x) \times \overrightarrow{x'}$

$(\texttt{tangent}\ ((\texttt{j*}\ f)\ (\texttt{bundle}\ x\ \overrightarrow{x'}))) = (((\mathcal{J}\ f)\ x)\ \overrightarrow{x'})$

A property

$x : \mathbb{R}^n$ $\qquad$ $\overrightarrow{x'} : \mathbb{R}^n$ $\qquad$ $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$

$((\mathcal{J}\ f)\ x)[i,j] = \frac{\partial f(x)[i]}{\partial x[j]}$ $\qquad$ $((\mathcal{J}\ f)\ x) : \mathbb{R}^{m \times n}$ $\qquad$ $((\mathcal{J}\ f)\ x) : \mathbb{R}^n \xrightarrow{L} \mathbb{R}^m$

$\texttt{(primal ((j}{\star}\ f\texttt{) (bundle } x\ \overrightarrow{x'}\texttt{)))} = \texttt{(}f\ x\texttt{)}$

$\texttt{(tangent ((j}{\star}\ f\texttt{) (bundle } x\ \overrightarrow{x'}\texttt{)))} = ((\mathcal{J}\ f)\ x) \times \overrightarrow{x'}$

$\texttt{(tangent ((j}{\star}\ f\texttt{) (bundle } x\ \overrightarrow{x'}\texttt{)))} = \texttt{(}((\mathcal{J}\ f)\ x)\ \overrightarrow{x'}\texttt{)}$

$\texttt{((j}{\star}\ f\texttt{) } x\texttt{)} = \texttt{(bundle (}f\texttt{ (primal } x\texttt{)) (}((\mathcal{J}\ f)\texttt{ (primal } x\texttt{)}) \texttt{ (tangent } x\texttt{)))}$

A property

$x : \mathbb{R}^n$ $\quad \overrightarrow{x'} : \mathbb{R}^n$ $\quad f : \mathbb{R}^n \rightarrow \mathbb{R}^m$

$((\mathcal{J}\, f)\ x)[i,j] = \frac{\partial f(x)[i]}{\partial x[j]}$ $\quad ((\mathcal{J}\, f)\ x) : \mathbb{R}^{m\times n}$ $\quad ((\mathcal{J}\, f)\ x) : \mathbb{R}^n \xrightarrow{L} \mathbb{R}^m$

(primal ((j* f) (bundle x $\overrightarrow{x'}$))) $= (f\ x)$

(tangent ((j* f) (bundle x $\overrightarrow{x'}$))) $= ((\mathcal{J}\, f)\ x) \times \overrightarrow{x'}$

(tangent ((j* f) (bundle x $\overrightarrow{x'}$))) $= (((\mathcal{J}\, f)\ x)\ \ \overrightarrow{x'})$

((j* f) x) $=$ (bundle (f (primal x)) $(((\mathcal{J}\, f)$ (primal x)$)$ (tangent x)))

rearrangement function: $(\forall i)(\exists j)(f\ x)[i] = x[j]$

A property

$x : \mathbb{R}^n$ $\quad \overrightarrow{x'} : \mathbb{R}^n$ $\quad f : \mathbb{R}^n \rightarrow \mathbb{R}^m$

$((\mathcal{J}\, f)\ x)[i,j] = \frac{\partial f(x)[i]}{\partial x[j]}$ $\quad ((\mathcal{J}\, f)\ x) : \mathbb{R}^{m\times n}$ $\quad ((\mathcal{J}\, f)\ x) : \mathbb{R}^n \xrightarrow{L} \mathbb{R}^m$

$\texttt{(primal ((j* } f\texttt{) (bundle } x\ \overrightarrow{x'}\texttt{)))} = (f\ \ x)$

$\texttt{(tangent ((j* } f\texttt{) (bundle } x\ \overrightarrow{x'}\texttt{)))} = ((\mathcal{J}\, f)\ x) \times \overrightarrow{x'}$

$\texttt{(tangent ((j* } f\texttt{) (bundle } x\ \overrightarrow{x'}\texttt{)))} = (((\mathcal{J}\, f)\ x)\ \ \overrightarrow{x'}\texttt{)}$

$\texttt{((j* } f\texttt{) } x\texttt{)} = \texttt{(bundle } (f\ \texttt{(primal } x\texttt{))}\ (((\mathcal{J}\, f)\ \texttt{(primal } x\texttt{)})\ \texttt{(tangent } x\texttt{)))}$

rearrangement function: $(\forall i)(\exists j)(f\ \ x)[i] = x[j]$

$f : \mathbb{R}^n \xrightarrow{L} \mathbb{R}^m$

A property

$x : \mathbb{R}^n$ $\qquad \overrightarrow{x'} : \mathbb{R}^n$ $\qquad f : \mathbb{R}^n \rightarrow \mathbb{R}^m$

$((\mathcal{J}\, f)\ x)[i,j] = \frac{\partial f(x)[i]}{\partial x[j]}$ $\qquad ((\mathcal{J}\, f)\ x) : \mathbb{R}^{m\times n}$ $\qquad ((\mathcal{J}\, f)\ x) : \mathbb{R}^n \xrightarrow{L} \mathbb{R}^m$

(primal ((j* f) (bundle x $\overrightarrow{x'}$))) $= (f\ x)$

(tangent ((j* f) (bundle x $\overrightarrow{x'}$))) $= ((\mathcal{J}\, f)\ x) \times \overrightarrow{x'}$

(tangent ((j* f) (bundle x $\overrightarrow{x'}$))) $= (((\mathcal{J}\, f)\ x)\ \overrightarrow{x'})$

((j* f) x) $=$ (bundle (f (primal x)) $(((\mathcal{J}\, f)$ (primal x)$)$ (tangent x)))

rearrangement function: $(\forall i)(\exists j)(f\ x)[i] = x[j]$

$f : \mathbb{R}^n \xrightarrow{L} \mathbb{R}^m$

$0/1$ matrix, every row has exactly one 1

A property

$x : \mathbb{R}^n$ $\quad$ $\overrightarrow{x'} : \mathbb{R}^n$ $\quad$ $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$

$((\mathcal{J}\ f)\ x)[i,j] = \frac{\partial f(x)[i]}{\partial x[j]}$ $\quad$ $((\mathcal{J}\ f)\ x) : \mathbb{R}^{m \times n}$ $\quad$ $((\mathcal{J}\ f)\ x) : \mathbb{R}^n \xrightarrow{L} \mathbb{R}^m$

$\texttt{(primal ((j* } f\texttt{) (bundle } x\ \overrightarrow{x'}\texttt{)))} = \texttt{(}f\ x\texttt{)}$

$\texttt{(tangent ((j* } f\texttt{) (bundle } x\ \overrightarrow{x'}\texttt{)))} = ((\mathcal{J}\ f)\ x) \times \overrightarrow{x'}$

$\texttt{(tangent ((j* } f\texttt{) (bundle } x\ \overrightarrow{x'}\texttt{)))} = \texttt{(}((\mathcal{J}\ f)\ x)\ \overrightarrow{x'}\texttt{)}$

$\texttt{((j* } f\texttt{) } x\texttt{)} = \texttt{(bundle (}f\texttt{ (primal } x\texttt{)) (}((\mathcal{J}\ f)\texttt{ (primal } x\texttt{)) (tangent } x\texttt{)))}$

rearrangement function: $(\forall i)(\exists j)(f\ x)[i] = x[j]$

$f : \mathbb{R}^n \xrightarrow{L} \mathbb{R}^m$

$0/1$ matrix, every row has exactly one 1

$((\mathcal{J}\ f)\ x)$

A property

$x : \mathbb{R}^n$ $\quad$ $\overleftarrow{x'} : \mathbb{R}^n$ $\quad$ $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$

$((\mathcal{J}\, f)\ x)[i,j] = \frac{\partial f(x)[i]}{\partial x[j]}$ $\quad$ $((\mathcal{J}\, f)\ x) : \mathbb{R}^{m \times n}$ $\quad$ $((\mathcal{J}\, f)\ x) : \mathbb{R}^n \xrightarrow{L} \mathbb{R}^m$

$\texttt{(primal ((j* } f\texttt{) (bundle } x\ \overleftarrow{x'}\texttt{)))} = (f\ \ x)$

$\texttt{(tangent ((j* } f\texttt{) (bundle } x\ \overleftarrow{x'}\texttt{)))} = ((\mathcal{J}\, f)\ x) \times \overleftarrow{x'}$

$\texttt{(tangent ((j* } f\texttt{) (bundle } x\ \overleftarrow{x'}\texttt{)))} = (((\mathcal{J}\, f)\ x)\ \ \overleftarrow{x'}\texttt{)}$

$\texttt{((j* } f\texttt{) } x\texttt{)} = \texttt{(bundle } (f\ \texttt{(primal } x\texttt{))}\ (((\mathcal{J}\, f)\ \texttt{(primal } x\texttt{)})\ \texttt{(tangent } x\texttt{)))}$

rearrangement function: $(\forall i)(\exists j)(f\ x)[i] = x[j]$

$f : \mathbb{R}^n \xrightarrow{L} \mathbb{R}^m$

$0/1$ matrix, every row has exactly one 1

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

A property

$x : \mathbb{R}^n$ $\qquad \overrightarrow{x'} : \mathbb{R}^n$ $\qquad f : \mathbb{R}^n \rightarrow \mathbb{R}^m$

$((\mathcal{J}\, f)\ x)[i,j] = \frac{\partial f(x)[i]}{\partial x[j]}$ $\qquad ((\mathcal{J}\, f)\ x) : \mathbb{R}^{m\times n}$ $\qquad ((\mathcal{J}\, f)\ x) : \mathbb{R}^n \xrightarrow{L} \mathbb{R}^m$

$\texttt{(primal ((j}\star\ f\texttt{) (bundle}\ x\ \overrightarrow{x'}\texttt{)))} = (f\ \ x)$

$\texttt{(tangent ((j}\star\ f\texttt{) (bundle}\ x\ \overrightarrow{x'}\texttt{)))} = ((\mathcal{J}\, f)\ x) \times \overrightarrow{x'}$

$\texttt{(tangent ((j}\star\ f\texttt{) (bundle}\ x\ \overrightarrow{x'}\texttt{)))} = (((\mathcal{J}\, f)\ x)\ \ \overrightarrow{x'}\texttt{)}$

$\texttt{((j}\star\ f\texttt{)}\ x\texttt{)} = \texttt{(bundle}\ (f\ \texttt{(primal}\ x\texttt{))}\ (((\mathcal{J}\, f)\ \texttt{(primal}\ x\texttt{))}\ \texttt{(tangent}\ x\texttt{)))}$

rearrangement function: $(\forall i)(\exists j)(f\ x)[i] = x[j]$

$f : \mathbb{R}^n \xrightarrow{L} \mathbb{R}^m$

$0/1$ matrix, every row has exactly one 1

$$((\mathcal{J}\, f)\ x) = \frac{\partial f(x)[i]}{\partial x[j]} = \begin{cases} 1 & \text{when } (f\ x)[i] = x[j] \\ 0 & \text{otherwise} \end{cases}$$

A property

$x : \mathbb{R}^n \qquad \overleftarrow{x'} : \mathbb{R}^n \qquad f : \mathbb{R}^n \to \mathbb{R}^m$

$((\mathcal{J}\, f)\; x)[i,j] = \frac{\partial f(x)[i]}{\partial x[j]} \qquad ((\mathcal{J}\, f)\; x) : \mathbb{R}^{m \times n} \qquad ((\mathcal{J}\, f)\; x) : \mathbb{R}^n \xrightarrow{L} \mathbb{R}^m$

$\texttt{(primal ((j* } f\texttt{) (bundle } x\ \overleftarrow{x'}\texttt{)))} = (f\;\; x)$

$\texttt{(tangent ((j* } f\texttt{) (bundle } x\ \overleftarrow{x'}\texttt{)))} = ((\mathcal{J}\, f)\; x) \times \overleftarrow{x'}$

$\texttt{(tangent ((j* } f\texttt{) (bundle } x\ \overleftarrow{x'}\texttt{)))} = (((\mathcal{J}\, f)\; x)\;\; \overleftarrow{x'})$

$\texttt{((j* } f\texttt{) } x\texttt{)} = \texttt{(bundle (}f\texttt{ (primal } x\texttt{)) (}((\mathcal{J}\, f)\texttt{ (primal } x\texttt{))}\texttt{ (tangent } x\texttt{)))}$

rearrangement function: $(\forall i)(\exists j)(f\;\; x)[i] = x[j]$

$f : \mathbb{R}^n \xrightarrow{L} \mathbb{R}^m$

0/1 matrix, every row has exactly one 1

$$((\mathcal{J}\, f)\; x) = \frac{\partial f(x)[i]}{\partial x[j]} = \begin{cases} 1 & \text{when } (f\;\; x)[i] = x[j] \\ 0 & \text{otherwise} \end{cases} = f$$

A property

$x : \mathbb{R}^n$ $\quad$ $\overrightarrow{x'} : \mathbb{R}^n$ $\quad$ $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$

$((\mathcal{J}\, f)\; x)[i,j] = \frac{\partial f(x)[i]}{\partial x[j]}$ $\quad$ $((\mathcal{J}\, f)\; x) : \mathbb{R}^{m \times n}$ $\quad$ $((\mathcal{J}\, f)\; x) : \mathbb{R}^n \xrightarrow{L} \mathbb{R}^m$

(primal ((j* f) (bundle x $\overrightarrow{x'}$))) $= (f\;\; x)$

(tangent ((j* f) (bundle x $\overrightarrow{x'}$))) $= ((\mathcal{J}\, f)\; x) \times \overrightarrow{x'}$

(tangent ((j* f) (bundle x $\overrightarrow{x'}$))) $= (((\mathcal{J}\, f)\; x)\;\; \overrightarrow{x'})$

((j* f) x) $=$ (bundle (f (primal x)) ((($\mathcal{J}\, f$) (primal x)) (tangent x)))

rearrangement function: $(\forall i)(\exists j)(f\;\; x)[i] = x[j]$

$f : \mathbb{R}^n \xrightarrow{L} \mathbb{R}^m$

0/1 matrix, every row has exactly one 1

$$((\mathcal{J}\, f)\; x) = \frac{\partial f(x)[i]}{\partial x[j]} = \begin{cases} 1 & \text{when } (f\;\; x)[i] = x[j] \\ 0 & \text{otherwise} \end{cases} = f$$

when f is a rearrangement function

((j* f) x) $=$ (bundle (f (primal x)) (f (tangent x)))

A property

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

$((\mathcal{J}\, f)\ x)[i,j] = \frac{\partial f(x)[i]}{\partial x[j]}$ $((\mathcal{J}\, f)\ x) : \mathbb{R}^{m\times n}$ $((\mathcal{J}\, f)\ x) : \mathbb{R}^n \xrightarrow{L} \mathbb{R}^m$

(primal ((j* f) (bundle x $\overrightarrow{x'}$))) $= (f\ x)$

(tangent ((j* f) (bundle x $\overrightarrow{x'}$))) $= ((\mathcal{J}\, f)\ x) \times \overrightarrow{x'}$

(tangent ((j* f) (bundle x $\overrightarrow{x'}$))) $= (((\mathcal{J}\, f)\ x)\ \ \overrightarrow{x'})$

((j* f) x) = (bundle (f (primal x)) $(((\mathcal{J}\, f)$ (primal x)$)$ (tangent x)))

rearrangement function: $(\forall i)(\exists j)(f\ x)[i] = x[j]$

$f : \mathbb{R}^n \xrightarrow{L} \mathbb{R}^m$

$0/1$ matrix, every row has exactly one 1

$$((\mathcal{J}\, f)\ x) = \frac{\partial f(x)[i]}{\partial x[j]} = \begin{cases} 1 & \text{when } (f\ x)[i] = x[j] \\ 0 & \text{otherwise} \end{cases} = f$$

when f is a rearrangement function
((j* f) x) = (bundle (f (primal x)) (f (tangent x)))

A property

$x : \tau_1$ $\qquad \overrightarrow{x'} : \tau_1$ $\qquad f : \tau_1 \rightarrow \tau_2$

$((\mathcal{J}\, f)\, x)[i,j] = \frac{\partial f(x)[i]}{\partial x[j]}$ $\qquad ((\mathcal{J}\, f)\, x) : \tau_1 \xrightarrow{L} \tau_2$

`(primal ((j*` f`) (bundle` x $\overrightarrow{x'}$`)))` $= ($`f` $x)$

`(tangent ((j*` f`) (bundle` x $\overrightarrow{x'}$`)))` $= ((\mathcal{J}\, f)\, x) \times \overrightarrow{x'}$

`(tangent ((j*` f`) (bundle` x $\overrightarrow{x'}$`)))` $= (((\mathcal{J}\, f)\, x)\ \overrightarrow{x'})$

`((j*` f`)` x`) = (bundle (`f `(primal` x`)) (`$((\mathcal{J}\, f)$ `(primal` x`))` `(tangent` x`)))`

rearrangement function: $(\forall i)(\exists j)(f\ x)[i] = x[j]$

$f : \tau_1 \xrightarrow{L} \tau_2$

0/1 matrix, every row has exactly one 1

$$((\mathcal{J}\, f)\, x) = \frac{\partial f(x)[i]}{\partial x[j]} = \begin{cases} 1 & \text{when } (f\ x)[i] = x[j] \\ 0 & \text{otherwise} \end{cases} = f$$

when f is a rearrangement function

`((j*` f`)` x`) = (bundle (`f `(primal` x`)) (`f `(tangent` x`)))`

A property

$x : \tau_1$ $\overrightarrow{x'} : \tau_1$ $f : \tau_1 \rightarrow \tau_2$

$((\mathcal{J}\, f)\; x)[i,j] = \frac{\partial f(x)[i]}{\partial x[j]}$ $((\mathcal{J}\, f)\; x) : \tau_1 \xrightarrow{L} \tau_2$

$\texttt{(primal ((j* } f\texttt{) (bundle } x\ \overrightarrow{x'}\texttt{)))} = \texttt{(}f\ x\texttt{)}$

$\texttt{(tangent ((j* } f\texttt{) (bundle } x\ \overrightarrow{x'}\texttt{)))} = ((\mathcal{J}\, f)\; x) \times \overrightarrow{x'}$

$\texttt{(tangent ((j* } f\texttt{) (bundle } x\ \overrightarrow{x'}\texttt{)))} = (((\mathcal{J}\, f)\; x)\ \overrightarrow{x'}\texttt{)}$

$\texttt{((j* } f\texttt{) } x\texttt{)} = \texttt{(bundle (}f\texttt{ (primal } x\texttt{)) }(((\mathcal{J}\, f)\texttt{ (primal } x\texttt{)) (tangent } x\texttt{)))}$

rearrangement function: $(\forall i)(\exists j)(f\; x)[i] = x[j]$

$f : \tau_1 \xrightarrow{L} \tau_2$

$0/1$ matrix, every row has exactly one 1

$$((\mathcal{J}\, f)\; x) = \frac{\partial f(x)[i]}{\partial x[j]} = \begin{cases} 1 & \text{when } (f\; x)[i] = x[j] \\ 0 & \text{otherwise} \end{cases} = f$$

when f is a rearrangement function

$\texttt{((j* } f\texttt{) } x\texttt{)} = \texttt{(bundle (}f\texttt{ (primal } x\texttt{)) (}f\texttt{ (tangent } x\texttt{)))}$

What is the tangent of `#t`?

What is the tangent of #t?

What if we take $\overrightarrow{\texttt{\#t}} = \texttt{\#f}$?

What is the tangent of #t?

What if we take $\overrightarrow{\texttt{\#t}} = \texttt{\#f}$?

when f is a rearrangement function

`((j*` f`)` x`)` $=$ `(bundle (`f `(primal` x`)) (`f `(tangent` x`)))`

What is the tangent of #t?

What if we take $\overrightarrow{\texttt{\#t}} = \texttt{\#f}$?

when f is a rearrangement function

$(\texttt{(j*}\ f\texttt{)}\ x\texttt{)} = \texttt{(bundle}\ (f\ \texttt{(primal}\ x\texttt{))}\ (f\ \texttt{(tangent}\ x\texttt{)))}$

$f : \texttt{(\#t}\ x\ y\texttt{)} \mapsto \texttt{(\#t}\ x\ y\texttt{)}$ but $f : \texttt{(\#f}\ x\ y\texttt{)} \mapsto \texttt{(\#f}\ y\ x\texttt{)}$

What is the tangent of `#t`?

What if we take $\overrightarrow{\texttt{\#t}} = \texttt{\#f}$?

when f is a rearrangement function

$\texttt{((j* } f\texttt{) } x\texttt{)} = \texttt{(bundle (}f\texttt{ (primal } x\texttt{)) (}f\texttt{ (tangent } x\texttt{)))}$

$f : \texttt{(\#t } x\ y\texttt{)} \mapsto \texttt{(\#t } x\ y\texttt{)}$ but $f : \texttt{(\#f } x\ y\texttt{)} \mapsto \texttt{(\#f } y\ x\texttt{)}$

f is a rearrangement function

What is the tangent of `#t`?

What if we take $\overrightarrow{\texttt{\#t}} = \texttt{\#f}$?

when f is a rearrangement function

`((j*` f`)` x`)` $=$ `(bundle (`f `(primal` x`)) (`f `(tangent` x`)))`

f : `(#t` x y`)` $\mapsto$ `(#t` x y`)` but f : `(#f` x y`)` $\mapsto$ `(#f` y x`)`

f is a rearrangement function

`((j*` f`) (bundle (#t` x y`) (#f` $\overrightarrow{x}$ $\overrightarrow{y}$`)))`

What is the tangent of `#t`?

What if we take $\overrightarrow{\texttt{\#t}} = \texttt{\#f}$?

when f is a rearrangement function

$(\texttt{(j*}\ f\texttt{)}\ x\texttt{)} = \texttt{(bundle}\ (f\ \texttt{(primal}\ x\texttt{))}\ (f\ \texttt{(tangent}\ x\texttt{)))}$

$f : \texttt{(\#t}\ x\ y\texttt{)} \mapsto \texttt{(\#t}\ x\ y\texttt{)}$ but $f : \texttt{(\#f}\ x\ y\texttt{)} \mapsto \texttt{(\#f}\ y\ x\texttt{)}$

f is a rearrangement function

$\texttt{((j*}\ f\texttt{)}\ \texttt{(bundle}\ \texttt{(\#t}\ x\ y\texttt{)}\ \texttt{(\#f}\ \overrightarrow{x}\ \overrightarrow{y}\texttt{)))}$

$= \texttt{(bundle}\ \texttt{(\#t}\ x\ y\texttt{)}\ \texttt{(\#f}\ \overrightarrow{y}\ \overrightarrow{x}\texttt{))}$

What is the tangent of `#t`?

What if we take $\overrightarrow{\texttt{\#t}} = \texttt{\#f}$?

when f is a rearrangement function

$\texttt{((j* } f\texttt{) } x\texttt{)} = \texttt{(bundle (}f\texttt{ (primal } x\texttt{)) (}f\texttt{ (tangent } x\texttt{)))}$

$f : \texttt{(\#t } x\ y\texttt{)} \mapsto \texttt{(\#t } x\ y\texttt{)}$ but $f : \texttt{(\#f } x\ y\texttt{)} \mapsto \texttt{(\#f } y\ x\texttt{)}$

f is a rearrangement function

$\texttt{((j* } f\texttt{) (bundle (\#t } x\ y\texttt{) (\#f } \overrightarrow{x}\ \overrightarrow{y}\texttt{)))}$

$= \texttt{(bundle (\#t } x\ y\texttt{) (\#f } \overrightarrow{y}\ \overrightarrow{x}\texttt{))}$

What is the tangent of `#t`?

What if we take $\overrightharpoon{\texttt{\#t}} = \texttt{\#f}$?

when f is a rearrangement function

$\texttt{((j* } f\texttt{) } x\texttt{)} = \texttt{(bundle (}f\texttt{ (primal } x\texttt{)) (}f\texttt{ (tangent } x\texttt{)))}$

$f: \texttt{(\#t } x\ y\texttt{)} \mapsto \texttt{(\#t } x\ y\texttt{)}$ but $f: \texttt{(\#f } x\ y\texttt{)} \mapsto \texttt{(\#f } y\ x\texttt{)}$

f is a rearrangement function

$\texttt{((j* } f\texttt{) (bundle (\#t } x\ y\texttt{) (\#f } \overrightharpoon{x}\ \overrightharpoon{y}\texttt{)))}$

$= \texttt{(bundle (\#t } x\ {\color{red}y}\texttt{) (\#f } \overrightharpoon{y}\ {\color{red}\overrightharpoon{x}}\texttt{))}$

What is the tangent of $\texttt{\#t}$?

What if we take $\overrightarrow{\texttt{\#t}} = \texttt{\#f}$?

when f is a rearrangement function

$\texttt{((j* }f\texttt{) }x\texttt{) = (bundle (}f\texttt{ (primal }x\texttt{)) (}f\texttt{ (tangent }x\texttt{)))}$

$f: \texttt{(\#t }x\ y\texttt{)} \mapsto \texttt{(\#t }x\ y\texttt{)}$ but $f: \texttt{(\#f }x\ y\texttt{)} \mapsto \texttt{(\#f }y\ x\texttt{)}$

f is a rearrangement function

$\texttt{((j* }f\texttt{) (bundle (\#t }x\ y\texttt{) (\#f }\overrightarrow{x}\ \overrightarrow{y}\texttt{)))}$

$= \texttt{(bundle (\#t }x\ y\texttt{) (\#f }\overrightarrow{y}\ \overrightarrow{x}\texttt{))}$

Problem avoided if we take $\overrightarrow{\texttt{\#t}} = \texttt{\#t}$

What is (j* j*)?

What is (j* j*)?

when f is a rearrangement function

((j* f) x) = (bundle (f (primal x)) (f (tangent x)))

What is `(j* j*)`?

when f is a rearrangement function

`((j*` f`)` x`)` $=$ `(bundle (`f `(primal` x`)) (`f `(tangent` x`)))`

`bundle`, `primal`, `tangent`, and `j*` are rearrangement functions

What is `(j* j*)`?

when f is a rearrangement function

`((j*` f`)` x`) = (bundle (`f `(primal` x`)) (`f `(tangent` x`)))`

`bundle`, `primal`, `tangent`, and `j*` are rearrangement functions

`((j* bundle)` x`) = (bundle (bundle (primal` x`)) (bundle (tangent` x`)))`

`((j* primal)` x`) = (bundle (primal (primal` x`)) (primal (tangent` x`)))`

`((j* tangent)` x`) = (bundle (tangent (primal` x`)) (tangent (tangent` x`)))`

`((j* j*)` x`) = (bundle (j* (primal` x`)) (j* (tangent` x`)))`

Modularity

$$\nabla\; f\; \mathbf{x} \quad \triangleq \quad \frac{\partial f(\mathbf{x})}{\partial x_1}, \ldots, \frac{\partial f(\mathbf{x})}{\partial x_n}$$

Modularity

$$\nabla\ f\ \mathbf{x} \quad \triangleq \quad \frac{\partial f(\mathbf{x})}{\partial x_1}, \ldots, \frac{\partial f(\mathbf{x})}{\partial x_n}$$

$$\textsc{GradientDescent}\ f\ \mathbf{x}_0 \quad \triangleq \quad \ldots \mathbf{x}_{i+1} := \ldots \nabla\ f\ \mathbf{x}_i \ldots$$

Modularity

$$\nabla\ f\ \mathbf{x} \quad \triangleq \quad \frac{\partial f(\mathbf{x})}{\partial x_1}, \ldots, \frac{\partial f(\mathbf{x})}{\partial x_n}$$

$$\textsc{GradientDescent}\ f\ \mathbf{x}_0 \quad \triangleq \quad \ldots \mathbf{x}_{i+1} := \ldots \nabla\ f\ \mathbf{x}_i \ldots$$

$$\text{argmin}\ f \quad \triangleq \quad \ldots \textsc{GradientDescent}\ f\ \mathbf{x}_0 \ldots$$

Modularity

$$\nabla\ f\ \mathbf{x} \triangleq \frac{\partial f(\mathbf{x})}{\partial x_1}, \ldots, \frac{\partial f(\mathbf{x})}{\partial x_n}$$

$$\textsc{GradientDescent}\ f\ \mathbf{x}_0 \triangleq \ldots \mathbf{x}_{i+1} := \ldots \nabla\ f\ \mathbf{x}_i \ldots$$

$$\text{argmin}\ f \triangleq \ldots \textsc{GradientDescent}\ f\ \mathbf{x}_0 \ldots$$

$$\textsc{NeutronFlux}\ r \triangleq \boxed{\textit{classified}}$$

Modularity

$$\nabla\ f\ \mathbf{x} \triangleq \frac{\partial f(\mathbf{x})}{\partial x_1}, \ldots, \frac{\partial f(\mathbf{x})}{\partial x_n}$$

$$\textsc{GradientDescent}\ f\ \mathbf{x}_0 \triangleq \ldots \mathbf{x}_{i+1} := \ldots \nabla\ f\ \mathbf{x}_i \ldots$$

$$\text{argmin}\ f \triangleq \ldots \textsc{GradientDescent}\ f\ \mathbf{x}_0 \ldots$$

$$\textsc{NeutronFlux}\ r \triangleq \boxed{\textit{classified}}$$

$$\textsc{Deviation}\ r \triangleq ((\textsc{NeutronFlux}\ r) - \textsc{NeutronFlux}_{\text{critical}})^2$$

Modularity

$$\nabla\ f\ \mathbf{x} \quad \triangleq \quad \frac{\partial f(\mathbf{x})}{\partial x_1}, \ldots, \frac{\partial f(\mathbf{x})}{\partial x_n}$$

$$\textsc{GradientDescent}\ f\ \mathbf{x}_0 \quad \triangleq \quad \ldots \mathbf{x}_{i+1} := \ldots \nabla\ f\ \mathbf{x}_i \ldots$$

$$\text{argmin}\ f \quad \triangleq \quad \ldots \textsc{GradientDescent}\ f\ \mathbf{x}_0 \ldots$$

$$\textsc{NeutronFlux}\ r \quad \triangleq \quad \boxed{\textit{classified}}$$

$$\textsc{Deviation}\ r \quad \triangleq \quad ((\textsc{NeutronFlux}\ r) - \textsc{NeutronFlux}_{\text{critical}})^2$$

$$r^* \quad \triangleq \quad \text{argmin}\ \textsc{Deviation}$$

Fermi, E. (1946). *The Development of the first chain reaction pile*. Proceedings of the American Philosophy Society, **90**:20–4.

Breaking Modularity

$$\nabla\, f\ \mathbf{x} \triangleq (\overrightarrow{f}\ \mathbf{x} \triangleright \overrightarrow{\mathbf{e}_1}), \ldots, (\overrightarrow{f}\ \mathbf{x} \triangleright \overrightarrow{\mathbf{e}_n})$$

$$\textsc{GradientDescent}\ f\ \mathbf{x}_0 \triangleq \ldots \mathbf{x}_{i+1} := \ldots \nabla\, f\ \mathbf{x}_i \ldots$$

$$\text{argmin}\ f \triangleq \ldots \textsc{GradientDescent}\ f\ \mathbf{x}_0 \ldots$$

$$\textsc{NeutronFlux}\ r \triangleq \boxed{\textit{classified}}$$

$$\textsc{Deviation}\ r \triangleq ((\textsc{NeutronFlux}\ r) - \textsc{NeutronFlux}_{\text{critical}})^2$$

$$r^* \triangleq \text{argmin}\ \textsc{Deviation}$$

Fermi, E. (1946). *The Development of the first chain reaction pile*. Proceedings of the American Philosophy Society, **90**:20–4.

Breaking Modularity

$\nabla\ \overrightarrow{f}\ \mathbf{x}$	$\triangleq$	$(\overrightarrow{f}\ \mathbf{x} \triangleright \overleftarrow{\mathbf{e}_1}), \ldots, (\overrightarrow{f}\ \mathbf{x} \triangleright \overleftarrow{\mathbf{e}_n})$
$\textsc{GradientDescent}\ f\ \mathbf{x}_0$	$\triangleq$	$\ldots \mathbf{x}_{i+1} := \ldots \nabla\ f\ \mathbf{x}_i \ldots$
$\text{argmin}\ f$	$\triangleq$	$\ldots \textsc{GradientDescent}\ f\ \mathbf{x}_0 \ldots$
$\textsc{NeutronFlux}\ r$	$\triangleq$	*classified*
$\textsc{Deviation}\ r$	$\triangleq$	$((\textsc{NeutronFlux}\ r) - \textsc{NeutronFlux}_{\text{critical}})^2$
r^*	$\triangleq$	$\text{argmin}\ \textsc{Deviation}$

Fermi, E. (1946). *The Development of the first chain reaction pile*. Proceedings of the American Philosophy Society, **90**:20–4.

Breaking Modularity

$\nabla\ \overrightarrow{f}\ \mathbf{x}$	$\triangleq$	$(\overrightarrow{f}\ \mathbf{x} \triangleright \overleftarrow{\mathbf{e}_1}), \ldots, (\overrightarrow{f}\ \mathbf{x} \triangleright \overleftarrow{\mathbf{e}_n})$
$\textsc{GradientDescent}\ f\ \mathbf{x}_0$	$\triangleq$	$\ldots \mathbf{x}_{i+1} := \ldots \nabla\ \overrightarrow{f}\ \mathbf{x}_i \ldots$
$\text{argmin}\ f$	$\triangleq$	$\ldots \textsc{GradientDescent}\ f\ \mathbf{x}_0 \ldots$
$\textsc{NeutronFlux}\ r$	$\triangleq$	*classified*
$\textsc{Deviation}\ r$	$\triangleq$	$((\textsc{NeutronFlux}\ r) - \textsc{NeutronFlux}_{\text{critical}})^2$
r^*	$\triangleq$	$\text{argmin}\ \textsc{Deviation}$

Fermi, E. (1946). *The Development of the first chain reaction pile*. Proceedings of the American Philosophy Society, **90**:20–4.

Breaking Modularity

$$\nabla\, \overrightarrow{f}\ \mathbf{x} \triangleq (\overrightarrow{f}\ \mathbf{x} \rhd \overrightarrow{\mathbf{e}_1}), \ldots, (\overrightarrow{f}\ \mathbf{x} \rhd \overrightarrow{\mathbf{e}_n})$$

$$\textsc{GradientDescent}\ \overrightarrow{f}\ \mathbf{x}_0 \triangleq \ldots \mathbf{x}_{i+1} := \ldots \nabla\, \overrightarrow{f}\ \mathbf{x}_i \ldots$$

$$\text{argmin}\ f \triangleq \ldots \textsc{GradientDescent}\ f\ \mathbf{x}_0 \ldots$$

$$\textsc{NeutronFlux}\ r \triangleq \boxed{\textit{classified}}$$

$$\textsc{Deviation}\ r \triangleq ((\textsc{NeutronFlux}\ r) - \textsc{NeutronFlux}_{\text{critical}})^2$$

$$r^* \triangleq \text{argmin}\ \textsc{Deviation}$$

Fermi, E. (1946). *The Development of the first chain reaction pile*. Proceedings of the American Philosophy Society, **90**:20–4.

Breaking Modularity

$\nabla \overrightarrow{f}\ \mathbf{x}$	$\triangleq$	$(\overrightarrow{f}\ \mathbf{x} \triangleright \overline{\mathbf{e}_1}), \ldots, (\overrightarrow{f}\ \mathbf{x} \triangleright \overline{\mathbf{e}_n})$
$\textsc{GradientDescent}\ \overrightarrow{f}\ \mathbf{x}_0$	$\triangleq$	$\ldots \mathbf{x}_{i+1} := \ldots \nabla \overrightarrow{f}\ \mathbf{x}_i \ldots$
$\text{argmin}\ f$	$\triangleq$	$\ldots \textsc{GradientDescent}\ \overrightarrow{f}\ \mathbf{x}_0 \ldots$
$\textsc{NeutronFlux}\ r$	$\triangleq$	*classified*
$\textsc{Deviation}\ r$	$\triangleq$	$((\textsc{NeutronFlux}\ r) - \textsc{NeutronFlux}_{\text{critical}})^2$
r^*	$\triangleq$	$\text{argmin}\ \textsc{Deviation}$

Fermi, E. (1946). *The Development of the first chain reaction pile*. Proceedings of the American Philosophy Society, **90**:20–4.

Breaking Modularity

$$\nabla\ \overrightarrow{f}\ \mathbf{x} \triangleq (\overrightarrow{f}\ \mathbf{x} \rhd \overleftarrow{\mathbf{e}_1}), \ldots, (\overrightarrow{f}\ \mathbf{x} \rhd \overleftarrow{\mathbf{e}_n})$$

$$\textsc{GradientDescent}\ \overrightarrow{f}\ \mathbf{x}_0 \triangleq \ldots \mathbf{x}_{i+1} := \ldots \nabla\ \overrightarrow{f}\ \mathbf{x}_i \ldots$$

$$\text{argmin}\ \overrightarrow{f} \triangleq \ldots \textsc{GradientDescent}\ \overrightarrow{f}\ \mathbf{x}_0 \ldots$$

$$\textsc{NeutronFlux}\ r \triangleq \boxed{\textit{classified}}$$

$$\textsc{Deviation}\ r \triangleq ((\textsc{NeutronFlux}\ r) - \textsc{NeutronFlux}_{\text{critical}})^2$$

$$r^* \triangleq \text{argmin}\ \textsc{Deviation}$$

Fermi, E. (1946). *The Development of the first chain reaction pile*. Proceedings of the American Philosophy Society, **90**:20–4.

Breaking Modularity

$$\nabla\ \overrightarrow{f}\ \mathbf{x} \triangleq (\overrightarrow{f}\ \mathbf{x} \rhd \overrightarrow{\mathbf{e}_1}), \ldots, (\overrightarrow{f}\ \mathbf{x} \rhd \overrightarrow{\mathbf{e}_n})$$

$$\textsc{GradientDescent}\ \overrightarrow{f}\ \mathbf{x}_0 \triangleq \ldots \mathbf{x}_{i+1} := \ldots \nabla\ \overrightarrow{f}\ \mathbf{x}_i \ldots$$

$$\text{argmin}\ \overrightarrow{f} \triangleq \ldots \textsc{GradientDescent}\ \overrightarrow{f}\ \mathbf{x}_0 \ldots$$

$$\textsc{NeutronFlux}\ r \triangleq \boxed{\textit{classified}}$$

$$\textsc{Deviation}\ r \triangleq ((\textsc{NeutronFlux}\ r) - \textsc{NeutronFlux}_{\text{critical}})^2$$

$$r^* \triangleq \text{argmin}\ \overrightarrow{\textsc{Deviation}}$$

Fermi, E. (1946). *The Development of the first chain reaction pile*. Proceedings of the American Philosophy Society, **90**:20–4.

Breaking Modularity

$\nabla\ \overrightarrow{f}\ \mathbf{x}$	$\triangleq$	$(\overrightarrow{f}\ \mathbf{x} \triangleright \overrightarrow{\mathbf{e}_1}), \ldots, (\overrightarrow{f}\ \mathbf{x} \triangleright \overrightarrow{\mathbf{e}_n})$
$\textsc{GradientDescent}\ \overrightarrow{f}\ \mathbf{x}_0$	$\triangleq$	$\ldots \mathbf{x}_{i+1} := \ldots \nabla\ \overrightarrow{f}\ \mathbf{x}_i \ldots$
$\text{argmin}\ \overrightarrow{f}$	$\triangleq$	$\ldots \textsc{GradientDescent}\ \overrightarrow{f}\ \mathbf{x}_0 \ldots$
$\textsc{NeutronFlux}\ r$	$\triangleq$	*classified*
$\textsc{Deviation}\ r$	$\triangleq$	$((\textsc{NeutronFlux}\ r) - \textsc{NeutronFlux}_{\text{critical}})^2$
$\textsc{Deviation}$	$\overset{\textsc{ADIFOR}}{\rightsquigarrow}$	$\overrightarrow{\textsc{Deviation}}$
r^*	$\triangleq$	$\text{argmin}\ \overrightarrow{\textsc{Deviation}}$

Fermi, E. (1946). *The Development of the first chain reaction pile*. Proceedings of the American Philosophy Society, **90**:20–4.

Breaking Modularity

$$\nabla\ \overrightarrow{f}\ \mathbf{x} \triangleq (\overrightarrow{f}\ \mathbf{x} \rhd \overrightarrow{\mathbf{e}_1}), \ldots, (\overrightarrow{f}\ \mathbf{x} \rhd \overrightarrow{\mathbf{e}_n})$$

$$\textsc{GradientDescent}\ \overrightarrow{f}\ \mathbf{x}_0 \triangleq \ldots \mathbf{x}_{i+1} := \ldots \nabla\ \overrightarrow{f}\ \mathbf{x}_i \ldots$$

$$\text{argmin}\ \overrightarrow{f} \triangleq \ldots \textsc{GradientDescent}\ \overrightarrow{f}\ \mathbf{x}_0 \ldots$$

$$\textsc{NeutronFlux}\ r \triangleq \boxed{\textit{classified}}$$

$$\textsc{NeutronFlux} \overset{\text{ADIFOR}}{\rightsquigarrow} \overrightarrow{\textsc{NeutronFlux}}$$

$$\textsc{Deviation}\ r \triangleq ((\textsc{NeutronFlux}\ r) - \textsc{NeutronFlux}_{\text{critical}})^2$$

$$\textsc{Deviation} \overset{\text{ADIFOR}}{\rightsquigarrow} \overrightarrow{\textsc{Deviation}}$$

$$r^* \triangleq \text{argmin}\ \overrightarrow{\textsc{Deviation}}$$

Fermi, E. (1946). *The Development of the first chain reaction pile*. Proceedings of the American Philosophy Society, **90**:20–4.

Breaking Modularity

$$\nabla\ \overrightarrow{f}\ \mathbf{x} \triangleq (\overrightarrow{f}\ \mathbf{x} \rhd \overrightarrow{\mathbf{e}_1}), \ldots, (\overrightarrow{f}\ \mathbf{x} \rhd \overrightarrow{\mathbf{e}_n})$$

$$\textsc{GradientDescent}\ \overrightarrow{f}\ \mathbf{x}_0 \triangleq \ldots \mathbf{x}_{i+1} := \ldots \nabla\ \overrightarrow{f}\ \mathbf{x}_i \ldots$$

$$\text{argmin}\ \overrightarrow{f} \triangleq \ldots \textsc{GradientDescent}\ \overrightarrow{f}\ \mathbf{x}_0 \ldots$$

$$\textsc{NeutronFlux}\ \mathbf{r} \triangleq \boxed{\textit{classified}}$$

$$\textsc{NeutronFlux} \overset{\text{ADIFOR}}{\rightsquigarrow} \overrightarrow{\textsc{NeutronFlux}}$$

$$\textsc{Deviation}\ \mathbf{r} \triangleq ((\textsc{NeutronFlux}\ \mathbf{r}) - \textsc{NeutronFlux}_{\text{critical}})^2$$

$$\textsc{Deviation} \overset{\text{ADIFOR}}{\rightsquigarrow} \overrightarrow{\textsc{Deviation}}$$

$$\mathbf{r}^* \triangleq \text{argmin}\ \overrightarrow{\textsc{Deviation}}$$

Fermi, E. (1946). *The Development of the first chain reaction pile*. Proceedings of the American Philosophy Society, **90**:20–4.

Breaking Modularity

$$\nabla\ \overrightarrow{f}\ \mathbf{x} \triangleq \ldots \overleftarrow{f}\ \mathbf{x} \ldots$$

$$\textsc{GradientDescent}\ \overrightarrow{f}\ \mathbf{x}_0 \triangleq \ldots \mathbf{x}_{i+1} := \ldots \nabla\ \overrightarrow{f}\ \mathbf{x}_i \ldots$$

$$\text{argmin}\ \overrightarrow{f} \triangleq \ldots \textsc{GradientDescent}\ \overrightarrow{f}\ \mathbf{x}_0 \ldots$$

$$\textsc{NeutronFlux}\ \mathbf{r} \triangleq \boxed{\textit{classified}}$$

$$\textsc{NeutronFlux} \overset{\textsc{adifor}}{\leadsto} \overrightarrow{\textsc{NeutronFlux}}$$

$$\textsc{Deviation}\ \mathbf{r} \triangleq ((\textsc{NeutronFlux}\ \mathbf{r}) - \textsc{NeutronFlux}_{\text{critical}})^2$$

$$\textsc{Deviation} \overset{\textsc{adifor}}{\leadsto} \overrightarrow{\textsc{Deviation}}$$

$$\mathbf{r}^* \triangleq \text{argmin}\ \overrightarrow{\textsc{Deviation}}$$

Fermi, E. (1946). *The Development of the first chain reaction pile*. Proceedings of the American Philosophy Society, **90**:20–4.

Breaking Modularity

$\nabla\ \overleftarrow{f}\ \mathbf{x}$	$\triangleq$	$\ldots\ \overleftarrow{f}\ \mathbf{x}\ldots$
$\textsc{GradientDescent}\ \overrightarrow{f}\ \mathbf{x}_0$	$\triangleq$	$\ldots\mathbf{x}_{i+1} := \ldots\nabla\ \overrightarrow{f}\ \mathbf{x}_i\ldots$
$\text{argmin}\ \overrightarrow{f}$	$\triangleq$	$\ldots\textsc{GradientDescent}\ \overrightarrow{f}\ \mathbf{x}_0\ldots$
$\textsc{NeutronFlux}\ \mathbf{r}$	$\triangleq$	*classified*
$\textsc{NeutronFlux}$	$\overset{\text{ADIFOR}}{\rightsquigarrow}$	$\overrightarrow{\textsc{NeutronFlux}}$
$\textsc{Deviation}\ \mathbf{r}$	$\triangleq$	$((\textsc{NeutronFlux}\ \mathbf{r}) - \textsc{NeutronFlux}_{\text{critical}})^2$
$\textsc{Deviation}$	$\overset{\text{ADIFOR}}{\rightsquigarrow}$	$\overrightarrow{\textsc{Deviation}}$
$\mathbf{r}^*$	$\triangleq$	$\text{argmin}\ \overrightarrow{\textsc{Deviation}}$

Fermi, E. (1946). *The Development of the first chain reaction pile*. Proceedings of the American Philosophy Society, **90**:20–4.

Breaking Modularity

$$\nabla\ \overleftarrow{f}\ \mathbf{x} \triangleq \ldots \overleftarrow{f}\ \mathbf{x} \ldots$$

$$\textsc{GradientDescent}\ \overrightarrow{f}\ \mathbf{x}_0 \triangleq \ldots \mathbf{x}_{i+1} := \ldots \nabla\ \overleftarrow{f}\ \mathbf{x}_i \ldots$$

$$\text{argmin}\ \overrightarrow{f} \triangleq \ldots \textsc{GradientDescent}\ \overrightarrow{f}\ \mathbf{x}_0 \ldots$$

$$\textsc{NeutronFlux}\ \mathbf{r} \triangleq \boxed{\textit{classified}}$$

$$\textsc{NeutronFlux} \overset{\text{ADIFOR}}{\rightsquigarrow} \overrightarrow{\textsc{NeutronFlux}}$$

$$\textsc{Deviation}\ \mathbf{r} \triangleq ((\textsc{NeutronFlux}\ \mathbf{r}) - \textsc{NeutronFlux}_{\text{critical}})^2$$

$$\textsc{Deviation} \overset{\text{ADIFOR}}{\rightsquigarrow} \overrightarrow{\textsc{Deviation}}$$

$$\mathbf{r}^* \triangleq \text{argmin}\ \overrightarrow{\textsc{Deviation}}$$

Fermi, E. (1946). *The Development of the first chain reaction pile*. Proceedings of the American Philosophy Society, **90**:20–4.

Breaking Modularity

$$\nabla \overleftarrow{f}\ \mathbf{x} \triangleq \ldots \overleftarrow{f}\ \mathbf{x} \ldots$$

$$\textsc{GradientDescent}\ \overleftarrow{f}\ \mathbf{x}_0 \triangleq \ldots \mathbf{x}_{i+1} := \ldots \nabla \overleftarrow{f}\ \mathbf{x}_i \ldots$$

$$\text{argmin}\ \overrightarrow{f} \triangleq \ldots \textsc{GradientDescent}\ \overrightarrow{f}\ \mathbf{x}_0 \ldots$$

$$\textsc{NeutronFlux}\ \mathbf{r} \triangleq \boxed{\textit{classified}}$$

$$\textsc{NeutronFlux} \overset{\text{ADIFOR}}{\rightsquigarrow} \overrightarrow{\textsc{NeutronFlux}}$$

$$\textsc{Deviation}\ \mathbf{r} \triangleq ((\textsc{NeutronFlux}\ \mathbf{r}) - \textsc{NeutronFlux}_{\text{critical}})^2$$

$$\textsc{Deviation} \overset{\text{ADIFOR}}{\rightsquigarrow} \overrightarrow{\textsc{Deviation}}$$

$$\mathbf{r}^* \triangleq \text{argmin}\ \overrightarrow{\textsc{Deviation}}$$

Fermi, E. (1946). *The Development of the first chain reaction pile*. Proceedings of the American Philosophy Society, **90**:20–4.

Breaking Modularity

$$\nabla\ \overleftarrow{f}\ \mathbf{x} \triangleq \ldots \overleftarrow{f}\ \mathbf{x} \ldots$$

$$\textsc{GradientDescent}\ \overleftarrow{f}\ \mathbf{x}_0 \triangleq \ldots \mathbf{x}_{i+1} := \ldots \nabla\ \overleftarrow{f}\ \mathbf{x}_i \ldots$$

$$\text{argmin}\ \overrightarrow{f} \triangleq \ldots \textsc{GradientDescent}\ \overleftarrow{f}\ \mathbf{x}_0 \ldots$$

$$\textsc{NeutronFlux}\ \mathbf{r} \triangleq \boxed{\textit{classified}}$$

$$\textsc{NeutronFlux} \overset{\text{ADIFOR}}{\leadsto} \overrightarrow{\textsc{NeutronFlux}}$$

$$\textsc{Deviation}\ \mathbf{r} \triangleq ((\textsc{NeutronFlux}\ \mathbf{r}) - \textsc{NeutronFlux}_{\text{critical}})^2$$

$$\textsc{Deviation} \overset{\text{ADIFOR}}{\leadsto} \overrightarrow{\textsc{Deviation}}$$

$$\mathbf{r}^* \triangleq \text{argmin}\ \overrightarrow{\textsc{Deviation}}$$

Fermi, E. (1946). *The Development of the first chain reaction pile*. Proceedings of the American Philosophy Society, **90**:20–4.

Breaking Modularity

$$\nabla\ \overleftarrow{f}\ \mathbf{x} \triangleq \ldots \overleftarrow{f}\ \mathbf{x} \ldots$$

$$\textsc{GradientDescent}\ \overleftarrow{f}\ \mathbf{x}_0 \triangleq \ldots \mathbf{x}_{i+1} := \ldots \nabla\ \overleftarrow{f}\ \mathbf{x}_i \ldots$$

$$\text{argmin}\ \overleftarrow{f} \triangleq \ldots \textsc{GradientDescent}\ \overleftarrow{f}\ \mathbf{x}_0 \ldots$$

$$\textsc{NeutronFlux}\ \mathbf{r} \triangleq \boxed{\textit{classified}}$$

$$\textsc{NeutronFlux} \overset{\text{ADIFOR}}{\rightsquigarrow} \overrightarrow{\textsc{NeutronFlux}}$$

$$\textsc{Deviation}\ \mathbf{r} \triangleq ((\textsc{NeutronFlux}\ \mathbf{r}) - \textsc{NeutronFlux}_{\text{critical}})^2$$

$$\textsc{Deviation} \overset{\text{ADIFOR}}{\rightsquigarrow} \overrightarrow{\textsc{Deviation}}$$

$$\mathbf{r}^* \triangleq \text{argmin}\ \overrightarrow{\textsc{Deviation}}$$

Fermi, E. (1946). *The Development of the first chain reaction pile*. Proceedings of the American Philosophy Society, **90**:20–4.

Breaking Modularity

$$\nabla\ \overleftarrow{f}\ \mathbf{x} \triangleq \ldots\ \overleftarrow{f}\ \mathbf{x} \ldots$$

$$\textsc{GradientDescent}\ \overleftarrow{f}\ \mathbf{x}_0 \triangleq \ldots \mathbf{x}_{i+1} := \ldots \nabla\ \overleftarrow{f}\ \mathbf{x}_i \ldots$$

$$\text{argmin}\ \overleftarrow{f} \triangleq \ldots \textsc{GradientDescent}\ \overleftarrow{f}\ \mathbf{x}_0 \ldots$$

$$\textsc{NeutronFlux}\ \mathbf{r} \triangleq \boxed{\textit{classified}}$$

$$\textsc{NeutronFlux} \overset{\text{ADIFOR}}{\rightsquigarrow} \overrightarrow{\textsc{NeutronFlux}}$$

$$\textsc{Deviation}\ \mathbf{r} \triangleq ((\textsc{NeutronFlux}\ \mathbf{r}) - \textsc{NeutronFlux}_{\text{critical}})^2$$

$$\textsc{Deviation} \overset{\text{ADIFOR}}{\rightsquigarrow} \overrightarrow{\textsc{Deviation}}$$

$$\mathbf{r}^* \triangleq \text{argmin}\ \overleftarrow{\textsc{Deviation}}$$

Fermi, E. (1946). *The Development of the first chain reaction pile*. Proceedings of the American Philosophy Society, **90**:20–4.

Breaking Modularity

$\nabla \overleftarrow{f}\ \mathbf{x}$	$\triangleq$	$\ldots \overleftarrow{f}\ \mathbf{x} \ldots$
$\textsc{GradientDescent}\ \overleftarrow{f}\ \mathbf{x}_0$	$\triangleq$	$\ldots \mathbf{x}_{i+1} := \ldots \nabla \overleftarrow{f}\ \mathbf{x}_i \ldots$
$\text{argmin}\ \overleftarrow{f}$	$\triangleq$	$\ldots \textsc{GradientDescent}\ \overleftarrow{f}\ \mathbf{x}_0 \ldots$
$\textsc{NeutronFlux}\ \mathbf{r}$	$\triangleq$	*classified*
$\textsc{NeutronFlux}$	$\overset{\text{ADIFOR}}{\rightsquigarrow}$	$\overleftarrow{\textsc{NeutronFlux}}$
$\textsc{Deviation}\ \mathbf{r}$	$\triangleq$	$((\textsc{NeutronFlux}\ \mathbf{r}) - \textsc{NeutronFlux}_{\text{critical}})^2$
$\textsc{Deviation}$	$\overset{\text{ADIFOR}}{\rightsquigarrow}$	$\overleftarrow{\textsc{Deviation}}$
$\mathbf{r}^*$	$\triangleq$	$\text{argmin}\ \overleftarrow{\textsc{Deviation}}$

Fermi, E. (1946). *The Development of the first chain reaction pile*. Proceedings of the American Philosophy Society, **90**:20–4.

Breaking Modularity

$$\nabla\ \overleftarrow{f}\ \mathbf{x} \triangleq \ldots \overleftarrow{f}\ \mathbf{x} \ldots$$

$$\textsc{GradientDescent}\ \overleftarrow{f}\ \mathbf{x}_0 \triangleq \ldots \mathbf{x}_{i+1} := \ldots \nabla\ \overleftarrow{f}\ \mathbf{x}_i \ldots$$

$$\text{argmin}\ \overleftarrow{f} \triangleq \ldots \textsc{GradientDescent}\ \overleftarrow{f}\ \mathbf{x}_0 \ldots$$

$$\textsc{NeutronFlux}\ \mathbf{r} \triangleq \boxed{\textit{classified}}$$

$$\textsc{NeutronFlux} \overset{\textsc{Tapenade}}{\rightsquigarrow} \overleftarrow{\textsc{NeutronFlux}}$$

$$\textsc{Deviation}\ \mathbf{r} \triangleq ((\textsc{NeutronFlux}\ \mathbf{r}) - \textsc{NeutronFlux}_{\text{critical}})^2$$

$$\textsc{Deviation} \overset{\textsc{Tapenade}}{\rightsquigarrow} \overleftarrow{\textsc{Deviation}}$$

$$\mathbf{r}^* \triangleq \text{argmin}\ \overleftarrow{\textsc{Deviation}}$$

Fermi, E. (1946). *The Development of the first chain reaction pile*. Proceedings of the American Philosophy Society, **90**:20–4.

Breaking Modularity

$\nabla \overleftarrow{f}\ \mathbf{x} \quad \triangleq \quad \ldots \overleftarrow{f}\ \mathbf{x} \ldots$

$\textsc{GradientDescent}\ \overleftarrow{f}\ \mathbf{x}_0 \quad \triangleq \quad \ldots \mathbf{x}_{i+1} := \ldots \nabla \overleftarrow{f}\ \mathbf{x}_i \ldots$

$\textsc{NewtonsMethod}\ \overleftarrow{f}\ \mathbf{x}_0 \quad \triangleq \quad \ldots \mathbf{x}_{i+1} := \ldots \nabla \overleftarrow{f}\ \mathbf{x}_i \ldots \mathcal{H} f\ \mathbf{x}_i \ldots$

$\text{argmin}\ \overleftarrow{f} \quad \triangleq \quad \ldots \textsc{GradientDescent}\ \overleftarrow{f}\ \mathbf{x}_0 \ldots$

$\textsc{NeutronFlux}\ \mathbf{r} \quad \triangleq \quad \boxed{\textit{classified}}$

$\textsc{NeutronFlux} \quad \overset{\textsc{Tapenade}}{\rightsquigarrow} \quad \overleftarrow{\textsc{NeutronFlux}}$

$\textsc{Deviation}\ \mathbf{r} \quad \triangleq \quad ((\textsc{NeutronFlux}\ \mathbf{r}) - \textsc{NeutronFlux}_{\text{critical}})^2$

$\textsc{Deviation} \quad \overset{\textsc{Tapenade}}{\rightsquigarrow} \quad \overleftarrow{\textsc{Deviation}}$

$\mathbf{r}^* \quad \triangleq \quad \text{argmin}\ \overleftarrow{\textsc{Deviation}}$

Fermi, E. (1946). *The Development of the first chain reaction pile*. Proceedings of the American Philosophy Society, **90**:20–4.

Breaking Modularity

$$\nabla\ \overleftarrow{f}\ \mathbf{x} \triangleq \ldots\ \overleftarrow{f}\ \mathbf{x}\ldots$$

$$\textsc{GradientDescent}\ \overleftarrow{f}\ \mathbf{x}_0 \triangleq \ldots \mathbf{x}_{i+1} := \ldots \nabla\ \overleftarrow{f}\ \mathbf{x}_i \ldots$$

$$\textsc{NewtonsMethod}\ \overleftarrow{f}\ \mathbf{x}_0 \triangleq \ldots \mathbf{x}_{i+1} := \ldots \nabla\ \overleftarrow{f}\ \mathbf{x}_i \ldots \mathcal{H}\, f\ \mathbf{x}_i \ldots$$

$$\text{argmin}\ \overleftarrow{f} \triangleq \ldots \textsc{NewtonsMethod}\ \overleftarrow{f}\ \mathbf{x}_0 \ldots$$

$$\textsc{NeutronFlux}\ \mathbf{r} \triangleq \boxed{\textit{classified}}$$

$$\textsc{NeutronFlux} \overset{\textsc{Tapenade}}{\rightsquigarrow} \overleftarrow{\textsc{NeutronFlux}}$$

$$\textsc{Deviation}\ \mathbf{r} \triangleq ((\textsc{NeutronFlux}\ \mathbf{r}) - \textsc{NeutronFlux}_{\text{critical}})^2$$

$$\textsc{Deviation} \overset{\textsc{Tapenade}}{\rightsquigarrow} \overleftarrow{\textsc{Deviation}}$$

$$\mathbf{r}^* \triangleq \text{argmin}\ \overleftarrow{\textsc{Deviation}}$$

Fermi, E. (1946). *The Development of the first chain reaction pile*. Proceedings of the American Philosophy Society, **90**:20–4.

Breaking Modularity

$$\nabla \overleftarrow{f}\ \mathbf{x} \triangleq \ldots \overleftarrow{f}\ \mathbf{x} \ldots$$

$$\mathcal{H}\, f\ \mathbf{x} \triangleq$$

$$\textsc{GradientDescent}\ \overleftarrow{f}\ \mathbf{x}_0 \triangleq \ldots \mathbf{x}_{i+1} := \ldots \nabla \overleftarrow{f}\ \mathbf{x}_i \ldots$$

$$\textsc{NewtonsMethod}\ \overleftarrow{f}\ \mathbf{x}_0 \triangleq \ldots \mathbf{x}_{i+1} := \ldots \nabla \overleftarrow{f}\ \mathbf{x}_i \ldots \mathcal{H}\, f\ \mathbf{x}_i \ldots$$

$$\text{argmin}\ \overleftarrow{f} \triangleq \ldots \textsc{NewtonsMethod}\ \overleftarrow{f}\ \mathbf{x}_0 \ldots$$

$$\textsc{NeutronFlux}\ \mathbf{r} \triangleq \boxed{\textit{classified}}$$

$$\textsc{NeutronFlux} \overset{\textsc{Tapenade}}{\rightsquigarrow} \overleftarrow{\textsc{NeutronFlux}}$$

$$\textsc{Deviation}\ \mathbf{r} \triangleq ((\textsc{NeutronFlux}\ \mathbf{r}) - \textsc{NeutronFlux}_{\text{critical}})^2$$

$$\textsc{Deviation} \overset{\textsc{Tapenade}}{\rightsquigarrow} \overleftarrow{\textsc{Deviation}}$$

$$\mathbf{r}^* \triangleq \text{argmin}\ \overleftarrow{\textsc{Deviation}}$$

Fermi, E. (1946). *The Development of the first chain reaction pile*. Proceedings of the American Philosophy Society, **90**:20–4.

Breaking Modularity

$$\nabla\ \overleftarrow{f}\ \mathbf{x} \triangleq \ldots \overleftarrow{f}\ \mathbf{x} \ldots$$

$$\mathcal{H}\, f\ \mathbf{x} \triangleq \ldots \overleftrightarrow{f}\ \ldots \mathbf{x} \ldots$$

$$\textsc{GradientDescent}\ \overleftarrow{f}\ \mathbf{x}_0 \triangleq \ldots \mathbf{x}_{i+1} := \ldots \nabla\ \overleftarrow{f}\ \mathbf{x}_i \ldots$$

$$\textsc{NewtonsMethod}\ \overleftarrow{f}\ \mathbf{x}_0 \triangleq \ldots \mathbf{x}_{i+1} := \ldots \nabla\ \overleftarrow{f}\ \mathbf{x}_i \ldots \mathcal{H}\, f\ \mathbf{x}_i \ldots$$

$$\text{argmin}\ \overleftarrow{f} \triangleq \ldots \textsc{NewtonsMethod}\ \overleftarrow{f}\ \mathbf{x}_0 \ldots$$

$$\textsc{NeutronFlux}\ \mathbf{r} \triangleq \boxed{\textit{classified}}$$

$$\textsc{NeutronFlux} \overset{\textsc{Tapenade}}{\rightsquigarrow} \overleftarrow{\textsc{NeutronFlux}}$$

$$\textsc{Deviation}\ \mathbf{r} \triangleq ((\textsc{NeutronFlux}\ \mathbf{r}) - \textsc{NeutronFlux}_{\text{critical}})^2$$

$$\textsc{Deviation} \overset{\textsc{Tapenade}}{\rightsquigarrow} \overleftarrow{\textsc{Deviation}}$$

$$\mathbf{r}^* \triangleq \text{argmin}\ \overleftarrow{\textsc{Deviation}}$$

Fermi, E. (1946). *The Development of the first chain reaction pile*. Proceedings of the American Philosophy Society, **90**:20–4.

Breaking Modularity

$$\nabla \overleftarrow{f}\ \mathbf{x} \triangleq \ldots \overleftarrow{f}\ \mathbf{x} \ldots$$

$$\mathcal{H} \overleftrightarrows{f}\ \mathbf{x} \triangleq \ldots \overleftrightarrows{f} \ldots \mathbf{x} \ldots$$

$$\textsc{GradientDescent}\ \overleftarrow{f}\ \mathbf{x}_0 \triangleq \ldots \mathbf{x}_{i+1} := \ldots \nabla \overleftarrow{f}\ \mathbf{x}_i \ldots$$

$$\textsc{NewtonsMethod}\ \overleftarrow{f}\ \mathbf{x}_0 \triangleq \ldots \mathbf{x}_{i+1} := \ldots \nabla \overleftarrow{f}\ \mathbf{x}_i \ldots \mathcal{H} f\ \mathbf{x}_i \ldots$$

$$\text{argmin}\ \overleftarrow{f} \triangleq \ldots \textsc{NewtonsMethod}\ \overleftarrow{f}\ \mathbf{x}_0 \ldots$$

$$\textsc{NeutronFlux}\ \mathbf{r} \triangleq \boxed{\textit{classified}}$$

$$\textsc{NeutronFlux} \overset{\textsc{Tapenade}}{\rightsquigarrow} \overleftarrow{\textsc{NeutronFlux}}$$

$$\textsc{Deviation}\ \mathbf{r} \triangleq ((\textsc{NeutronFlux}\ \mathbf{r}) - \textsc{NeutronFlux}_{\text{critical}})^2$$

$$\textsc{Deviation} \overset{\textsc{Tapenade}}{\rightsquigarrow} \overleftarrow{\textsc{Deviation}}$$

$$\mathbf{r}^* \triangleq \text{argmin}\ \overleftarrow{\textsc{Deviation}}$$

Fermi, E. (1946). *The Development of the first chain reaction pile*. Proceedings of the American Philosophy Society, **90**:20–4.

Breaking Modularity

$$\nabla\ \overleftarrow{f}\ \mathbf{x} \triangleq \ldots \overleftarrow{f}\ \mathbf{x}\ldots$$

$$\mathcal{H}\ \overleftrightarrows{f}\ \mathbf{x} \triangleq \ldots \overleftrightarrows{f}\ \ldots\mathbf{x}\ldots$$

$$\textsc{GradientDescent}\ \overleftarrow{f}\ \mathbf{x}_0 \triangleq \ldots\mathbf{x}_{i+1} := \ldots\nabla\ \overleftarrow{f}\ \mathbf{x}_i\ldots$$

$$\textsc{NewtonsMethod}\ \overleftarrow{f}\ \mathbf{x}_0 \triangleq \ldots\mathbf{x}_{i+1} := \ldots\nabla\ \overleftarrow{f}\ \mathbf{x}_i\ldots\mathcal{H}\ \overleftrightarrows{f}\ \mathbf{x}_i\ldots$$

$$\operatorname{argmin}\ \overleftarrow{f} \triangleq \ldots\textsc{NewtonsMethod}\ \overleftarrow{f}\ \mathbf{x}_0\ldots$$

$$\textsc{NeutronFlux}\ \mathbf{r} \triangleq \boxed{\textit{classified}}$$

$$\textsc{NeutronFlux} \overset{\textsc{Tapenade}}{\rightsquigarrow} \overleftarrow{\textsc{NeutronFlux}}$$

$$\textsc{Deviation}\ \mathbf{r} \triangleq ((\textsc{NeutronFlux}\ \mathbf{r}) - \textsc{NeutronFlux}_{\text{critical}})^2$$

$$\textsc{Deviation} \overset{\textsc{Tapenade}}{\rightsquigarrow} \overleftarrow{\textsc{Deviation}}$$

$$\mathbf{r}^* \triangleq \operatorname{argmin}\ \overleftarrow{\textsc{Deviation}}$$

Fermi, E. (1946). *The Development of the first chain reaction pile*. Proceedings of the American Philosophy Society, **90**:20–4.

Breaking Modularity

$$\nabla\ \overleftarrow{f}\ \mathbf{x} \triangleq \ldots \overleftarrow{f}\ \mathbf{x}\ldots$$

$$\mathcal{H}\ \overrightleftharpoons{f}\ \mathbf{x} \triangleq \ldots \overrightleftharpoons{f}\ \ldots\mathbf{x}\ldots$$

$$\textsc{GradientDescent}\ \overleftarrow{f}\ \mathbf{x}_0 \triangleq \ldots\mathbf{x}_{i+1} := \ldots\nabla\ \overleftarrow{f}\ \mathbf{x}_i\ldots$$

$$\textsc{NewtonsMethod}\ \overleftarrow{f}\ \overrightleftharpoons{f}\ \mathbf{x}_0 \triangleq \ldots\mathbf{x}_{i+1} := \ldots\nabla\ \overleftarrow{f}\ \mathbf{x}_i\ldots\mathcal{H}\ \overrightleftharpoons{f}\ \mathbf{x}_i\ldots$$

$$\mathrm{argmin}\ \overleftarrow{f} \triangleq \ldots\textsc{NewtonsMethod}\ \overleftarrow{f}\ \mathbf{x}_0\ldots$$

$$\textsc{NeutronFlux}\ \mathbf{r} \triangleq \boxed{\textit{classified}}$$

$$\textsc{NeutronFlux} \overset{\textsc{Tapenade}}{\rightsquigarrow} \overleftarrow{\textsc{NeutronFlux}}$$

$$\textsc{Deviation}\ \mathbf{r} \triangleq ((\textsc{NeutronFlux}\ \mathbf{r}) - \textsc{NeutronFlux}_{\text{critical}})^2$$

$$\textsc{Deviation} \overset{\textsc{Tapenade}}{\rightsquigarrow} \overleftarrow{\textsc{Deviation}}$$

$$\mathbf{r}^* \triangleq \mathrm{argmin}\ \overleftarrow{\textsc{Deviation}}$$

Fermi, E. (1946). *The Development of the first chain reaction pile*. Proceedings of the American Philosophy Society, **90**:20–4.

Breaking Modularity

$\nabla\ \overleftarrow{f}\ \mathbf{x}$	$\triangleq$	$\ldots\ \overleftarrow{f}\ \mathbf{x}\ldots$
$\mathcal{H}\ \overleftrightarrows{f}\ \mathbf{x}$	$\triangleq$	$\ldots\ \overleftrightarrows{f}\ \ldots\mathbf{x}\ldots$
$\textsc{GradientDescent}\ \overleftarrow{f}\ \mathbf{x}_0$	$\triangleq$	$\ldots\mathbf{x}_{i+1} := \ldots\nabla\ \overleftarrow{f}\ \mathbf{x}_i\ldots$
$\textsc{NewtonsMethod}\ \overleftarrow{f}\ \overleftrightarrows{f}\ \mathbf{x}_0$	$\triangleq$	$\ldots\mathbf{x}_{i+1} := \ldots\nabla\ \overleftarrow{f}\ \mathbf{x}_i\ldots\mathcal{H}\ \overleftrightarrows{f}\ \mathbf{x}_i\ldots$
$\operatorname{argmin}\ \overleftarrow{f}$	$\triangleq$	$\ldots\textsc{NewtonsMethod}\ \overleftarrow{f}\ \overleftrightarrows{f}\ \mathbf{x}_0\ldots$
$\textsc{NeutronFlux}\ \mathbf{r}$	$\triangleq$	*classified*
$\textsc{NeutronFlux}$	$\overset{\textsc{Tapenade}}{\rightsquigarrow}$	$\overleftarrow{\textsc{NeutronFlux}}$
$\textsc{Deviation}\ \mathbf{r}$	$\triangleq$	$((\textsc{NeutronFlux}\ \mathbf{r}) - \textsc{NeutronFlux}_{\text{critical}})^2$
$\textsc{Deviation}$	$\overset{\textsc{Tapenade}}{\rightsquigarrow}$	$\overleftarrow{\textsc{Deviation}}$
$\mathbf{r}^*$	$\triangleq$	$\operatorname{argmin}\ \overleftarrow{\textsc{Deviation}}$

Fermi, E. (1946). *The Development of the first chain reaction pile*. Proceedings of the American Philosophy Society, **90**:20–4.

Breaking Modularity

$$\nabla \overleftarrow{f}\ \mathbf{x} \triangleq \ldots \overleftarrow{f}\ \mathbf{x} \ldots$$

$$\mathcal{H}\ \overleftrightarrows{f}\ \mathbf{x} \triangleq \ldots \overleftrightarrows{f}\ \ldots \mathbf{x} \ldots$$

$$\textsc{GradientDescent}\ \overleftarrow{f}\ \mathbf{x}_0 \triangleq \ldots \mathbf{x}_{i+1} := \ldots \nabla \overleftarrow{f}\ \mathbf{x}_i \ldots$$

$$\textsc{NewtonsMethod}\ \overleftarrow{f}\ \overleftrightarrows{f}\ \mathbf{x}_0 \triangleq \ldots \mathbf{x}_{i+1} := \ldots \nabla \overleftarrow{f}\ \mathbf{x}_i \ldots \mathcal{H}\ \overleftrightarrows{f}\ \mathbf{x}_i \ldots$$

$$\text{argmin}\ \overleftarrow{f}\ \overleftrightarrows{f} \triangleq \ldots \textsc{NewtonsMethod}\ \overleftarrow{f}\ \overleftrightarrows{f}\ \mathbf{x}_0 \ldots$$

$$\textsc{NeutronFlux}\ \mathbf{r} \triangleq \boxed{\textit{classified}}$$

$$\textsc{NeutronFlux} \overset{\textsc{Tapenade}}{\rightsquigarrow} \overleftarrow{\textsc{NeutronFlux}}$$

$$\textsc{Deviation}\ \mathbf{r} \triangleq ((\textsc{NeutronFlux}\ \mathbf{r}) - \textsc{NeutronFlux}_{\text{critical}})^2$$

$$\textsc{Deviation} \overset{\textsc{Tapenade}}{\rightsquigarrow} \overleftarrow{\textsc{Deviation}}$$

$$\mathbf{r}^* \triangleq \text{argmin}\ \overleftarrow{\textsc{Deviation}}$$

Fermi, E. (1946). *The Development of the first chain reaction pile*. Proceedings of the American Philosophy Society, **90**:20–4.

Breaking Modularity

$$\nabla\ \overleftarrow{f}\ \mathbf{x} \triangleq \ldots\ \overleftarrow{f}\ \mathbf{x}\ldots$$

$$\mathcal{H}\ \overset{\rightleftarrows}{f}\ \mathbf{x} \triangleq \ldots\ \overset{\rightleftarrows}{f}\ \ldots\mathbf{x}\ldots$$

$$\textsc{GradientDescent}\ \overleftarrow{f}\ \mathbf{x}_0 \triangleq \ldots\mathbf{x}_{i+1} := \ldots\nabla\ \overleftarrow{f}\ \mathbf{x}_i\ldots$$

$$\textsc{NewtonsMethod}\ \overleftarrow{f}\ \overset{\rightleftarrows}{f}\ \mathbf{x}_0 \triangleq \ldots\mathbf{x}_{i+1} := \ldots\nabla\ \overleftarrow{f}\ \mathbf{x}_i\ldots\mathcal{H}\ \overset{\rightleftarrows}{f}\ \mathbf{x}_i\ldots$$

$$\text{argmin}\ \overleftarrow{f}\ \overset{\rightleftarrows}{f} \triangleq \ldots\textsc{NewtonsMethod}\ \overleftarrow{f}\ \overset{\rightleftarrows}{f}\ \mathbf{x}_0\ldots$$

$$\textsc{NeutronFlux}\ \mathbf{r} \triangleq \boxed{\textit{classified}}$$

$$\textsc{NeutronFlux} \overset{\textsc{Tapenade}}{\rightsquigarrow} \overleftarrow{\textsc{NeutronFlux}}$$

$$\textsc{Deviation}\ \mathbf{r} \triangleq ((\textsc{NeutronFlux}\ \mathbf{r}) - \textsc{NeutronFlux}_{\text{critical}})^2$$

$$\textsc{Deviation} \overset{\textsc{Tapenade}}{\rightsquigarrow} \overleftarrow{\textsc{Deviation}}$$

$$\mathbf{r}^* \triangleq \text{argmin}\ \overleftarrow{\textsc{Deviation}}\ \overset{\rightleftarrows}{\textsc{Deviation}}$$

Fermi, E. (1946). *The Development of the first chain reaction pile*. Proceedings of the American Philosophy Society, **90**:20–4.

Breaking Modularity

$$
\begin{array}{lcl}
\nabla\ \overleftarrow{f}\ \mathbf{x} & \triangleq & \ldots\ \overleftarrow{f}\ \mathbf{x}\ldots \\
\mathcal{H}\ \overleftrightarrow{f}\ \mathbf{x} & \triangleq & \ldots\ \overleftrightarrow{f}\ \ldots\mathbf{x}\ldots \\
\textsc{GradientDescent}\ \overleftarrow{f}\ \mathbf{x}_0 & \triangleq & \ldots\mathbf{x}_{i+1} := \ldots\nabla\ \overleftarrow{f}\ \mathbf{x}_i\ldots \\
\textsc{NewtonsMethod}\ \overleftarrow{f}\ \overleftrightarrow{f}\ \mathbf{x}_0 & \triangleq & \ldots\mathbf{x}_{i+1} := \ldots\nabla\ \overleftarrow{f}\ \mathbf{x}_i\ldots\mathcal{H}\ \overleftrightarrow{f}\ \mathbf{x}_i\ldots \\
\operatorname{argmin}\ \overleftarrow{f}\ \overleftrightarrow{f} & \triangleq & \ldots\textsc{NewtonsMethod}\ \overleftarrow{f}\ \overleftrightarrow{f}\ \mathbf{x}_0\ldots \\
\textsc{NeutronFlux}\ \mathbf{r} & \triangleq & \boxed{\textit{classified}} \\
\textsc{NeutronFlux} & \overset{\textsc{Tapenade}}{\rightsquigarrow} & \overleftarrow{\textsc{NeutronFlux}} \\
\overleftarrow{\textsc{NeutronFlux}} & \overset{\textsc{Tapenade}}{\rightsquigarrow} & \overleftrightarrow{\textsc{NeutronFlux}} \\
\textsc{Deviation}\ \mathbf{r} & \triangleq & ((\textsc{NeutronFlux}\ \mathbf{r}) - \textsc{NeutronFlux}_{\text{critical}})^2 \\
\textsc{Deviation} & \overset{\textsc{Tapenade}}{\rightsquigarrow} & \overleftarrow{\textsc{Deviation}} \\
\overleftarrow{\textsc{Deviation}} & \overset{\textsc{Tapenade}}{\rightsquigarrow} & \overleftrightarrow{\textsc{Deviation}} \\
\mathbf{r}^* & \triangleq & \operatorname{argmin}\ \overleftarrow{\textsc{Deviation}}\ \overleftrightarrow{\textsc{Deviation}}
\end{array}
$$

Fermi, E. (1946). *The Development of the first chain reaction pile*. Proceedings of the American Philosophy Society, **90**:20–4.

Restoring Modularity

$$\nabla\ f\ \mathbf{x} \triangleq$$

$$\mathcal{H}\ f\ \mathbf{x} \triangleq$$

$$\textsc{GradientDescent}\ f\ \mathbf{x}_0 \triangleq \ldots \mathbf{x}_{i+1} := \ldots \nabla\ f\ \mathbf{x}_i \ldots$$

$$\textsc{NewtonsMethod}\ f\ \mathbf{x}_0 \triangleq \ldots \mathbf{x}_{i+1} := \ldots \nabla\ f\ \mathbf{x}_i \ldots \mathcal{H}\ f\ \mathbf{x}_i \ldots$$

$$\operatorname{argmin}\ f \triangleq \ldots \textsc{GradientDescent}\ f\ \mathbf{x}_0 \ldots$$

$$\textsc{NeutronFlux}\ \mathbf{r} \triangleq \boxed{\textit{classified}}$$

$$\textsc{Deviation}\ \mathbf{r} \triangleq ((\textsc{NeutronFlux}\ \mathbf{r}) - \textsc{NeutronFlux}_{\text{critical}})^2$$

$$\mathbf{r}^* \triangleq \operatorname{argmin}\ \textsc{Deviation}$$

Fermi, E. (1946). *The Development of the first chain reaction pile*. Proceedings of the American Philosophy Society, **90**:20–4.

Restoring Modularity

$$\nabla\ f\ \mathbf{x} \triangleq ((\overrightarrow{\mathcal{J}}\ f)\ \mathbf{x} \triangleright \overleftarrow{\mathbf{e}_1}), \ldots, ((\overrightarrow{\mathcal{J}}\ f)\ \mathbf{x} \triangleright \overleftarrow{\mathbf{e}_n})$$

$$\mathcal{H}\ f\ \mathbf{x} \triangleq$$

$$\textsc{GradientDescent}\ f\ \mathbf{x}_0 \triangleq \ldots \mathbf{x}_{i+1} := \ldots \nabla\ f\ \mathbf{x}_i \ldots$$

$$\textsc{NewtonsMethod}\ f\ \mathbf{x}_0 \triangleq \ldots \mathbf{x}_{i+1} := \ldots \nabla\ f\ \mathbf{x}_i \ldots \mathcal{H}\ f\ \mathbf{x}_i \ldots$$

$$\text{argmin}\ f \triangleq \ldots \textsc{GradientDescent}\ f\ \mathbf{x}_0 \ldots$$

$$\textsc{NeutronFlux}\ \mathbf{r} \triangleq \boxed{\textit{classified}}$$

$$\textsc{Deviation}\ \mathbf{r} \triangleq ((\textsc{NeutronFlux}\ \mathbf{r}) - \textsc{NeutronFlux}_{\text{critical}})^2$$

$$\mathbf{r}^* \triangleq \text{argmin}\ \textsc{Deviation}$$

Fermi, E. (1946). *The Development of the first chain reaction pile*. Proceedings of the American Philosophy Society, **90**:20–4.

Restoring Modularity

$$\nabla\, f\ \mathbf{x} \triangleq \ldots(\overleftarrow{\mathcal{J}}\, f)\ \mathbf{x}\ldots$$

$$\mathcal{H}\, f\ \mathbf{x} \triangleq$$

$$\textsc{GradientDescent}\ f\ \mathbf{x}_0 \triangleq \ldots\mathbf{x}_{i+1} := \ldots\nabla\, f\ \mathbf{x}_i\ldots$$

$$\textsc{NewtonsMethod}\ f\ \mathbf{x}_0 \triangleq \ldots\mathbf{x}_{i+1} := \ldots\nabla\, f\ \mathbf{x}_i\ldots\mathcal{H}\, f\ \mathbf{x}_i\ldots$$

$$\text{argmin}\ f \triangleq \ldots\textsc{GradientDescent}\ f\ \mathbf{x}_0\ldots$$

$$\textsc{NeutronFlux}\ \mathbf{r} \triangleq \boxed{\textit{classified}}$$

$$\textsc{Deviation}\ \mathbf{r} \triangleq ((\textsc{NeutronFlux}\ \mathbf{r}) - \textsc{NeutronFlux}_{\text{critical}})^2$$

$$\mathbf{r}^* \triangleq \text{argmin}\ \textsc{Deviation}$$

Fermi, E. (1946). *The Development of the first chain reaction pile*. Proceedings of the American Philosophy Society, **90**:20–4.

Restoring Modularity

$$\nabla\, f\, \mathbf{x} \quad \triangleq \quad \ldots (\overleftarrow{\mathcal{J}}\, f)\, \mathbf{x} \ldots$$

$$\mathcal{H}\, f\, \mathbf{x} \quad \triangleq \quad \ldots (\overrightarrow{\mathcal{J}}\, (\overleftarrow{\mathcal{J}}\, f)) \ldots \mathbf{x} \ldots$$

$$\textsc{GradientDescent}\, f\, \mathbf{x}_0 \quad \triangleq \quad \ldots \mathbf{x}_{i+1} := \ldots \nabla\, f\, \mathbf{x}_i \ldots$$

$$\textsc{NewtonsMethod}\, f\, \mathbf{x}_0 \quad \triangleq \quad \ldots \mathbf{x}_{i+1} := \ldots \nabla\, f\, \mathbf{x}_i \ldots \mathcal{H}\, f\, \mathbf{x}_i \ldots$$

$$\operatorname{argmin}\, f \quad \triangleq \quad \ldots \textsc{NewtonsMethod}\, f\, \mathbf{x}_0 \ldots$$

$$\textsc{NeutronFlux}\, \mathbf{r} \quad \triangleq \quad \boxed{\textit{classified}}$$

$$\textsc{Deviation}\, \mathbf{r} \quad \triangleq \quad ((\textsc{NeutronFlux}\, \mathbf{r}) - \textsc{NeutronFlux}_{\text{critical}})^2$$

$$\mathbf{r}^* \quad \triangleq \quad \operatorname{argmin}\, \textsc{Deviation}$$

Fermi, E. (1946). *The Development of the first chain reaction pile*. Proceedings of the American Philosophy Society, **90**:20–4.

Having your cake and eating it too

- Convenient

- Fast

Having your cake and eating it too

- Convenient
  - $\mathcal{D}$ formulated as a higher-order function in the language
- Fast

Having your cake and eating it too

- Convenient
  - $\mathcal{D}$ formulated as a higher-order function in the language
  - no arbitrary restrictions
    applies to all data types and constructs in the language, including code produced by $\mathcal{D}$ and even $\mathcal{D}$ itself
- Fast

Having your cake and eating it too

- Convenient
  - $\mathcal{D}$ formulated as a higher-order function in the language
  - no arbitrary restrictions
    applies to all data types and constructs in the language, including code produced by $\mathcal{D}$ and even $\mathcal{D}$ itself
  - higher-order derivatives
    ($\mathcal{D}$ ($\mathcal{D}$ f))
- Fast

Having your cake and eating it too

- Convenient
  - $\mathcal{D}$ formulated as a higher-order function in the language
  - no arbitrary restrictions
    applies to all data types and constructs in the language, including code produced by $\mathcal{D}$ and even $\mathcal{D}$ itself
  - higher-order derivatives
    `(`$\mathcal{D}$` (`$\mathcal{D}$` f))`
  - nesting
    `(`$\mathcal{D}$` (lambda (...) ... (`$\mathcal{D}$` (lambda (...) ...)) ...))`
- Fast

Having your cake and eating it too

- Convenient
  - $\mathcal{D}$ formulated as a higher-order function in the language
  - no arbitrary restrictions
    applies to all data types and constructs in the language, including code produced by $\mathcal{D}$ and even $\mathcal{D}$ itself
  - higher-order derivatives
    `(`$\mathcal{D}$` (`$\mathcal{D}$` f))`
  - nesting
    `(`$\mathcal{D}$` (lambda (...) ... (`$\mathcal{D}$` (lambda (...) ...)) ...))`
- Fast
  - $\mathcal{D}$ implemented by reflective transformation of environments and code associated with closures

Having your cake and eating it too

- Convenient
  - $\mathcal{D}$ formulated as a higher-order function in the language
  - no arbitrary restrictions
    applies to all data types and constructs in the language, including code produced by $\mathcal{D}$ and even $\mathcal{D}$ itself
  - higher-order derivatives
    `(𝒟 (𝒟 f))`
  - nesting
    `(𝒟 (lambda (...) ... (𝒟 (lambda (...) ...)) ...))`
- Fast
  - $\mathcal{D}$ implemented by reflective transformation of environments and code associated with closures
  - compile away reflection with partial evaluation implemented by flow analysis

Monovariant Flow Analysis: 0-CFA

```
(define (𝒟 f)
 ...)
```

Monovariant Flow Analysis: 0-CFA

```
(define (𝒟 f)
 ...)
```

(𝒟 (lambda (x) $2x^3$))

Monovariant Flow Analysis: 0-CFA

```
(define (𝒟 f:(λx 2x³))
 ...)
```

```
(𝒟 (lambda (x) 2x³))
```

Monovariant Flow Analysis: 0-CFA

`(define (`$\mathcal{D}$` f:`$(\lambda x\ 2x^3)$`)`
` ...:`$(\lambda x\ 6x^2)$`)`

`(`$\mathcal{D}$` (lambda (x) `$2x^3$`))`

Monovariant Flow Analysis: 0-CFA

```
(define (𝒟 f:(λx 2x³))
 ...:(λx 6x²))
```

(𝒟 (lambda (x) $2x^3$)):$(\lambda x\ 6x^2)$

Monovariant Flow Analysis: 0-CFA

(define ($\mathcal{D}$ f:$(\lambda x\ 2x^3)$)
...:$(\lambda x\ 6x^2)$)

($\mathcal{D}$ (lambda (x) $2x^3$)):$(\lambda x\ 6x^2)$

($\mathcal{D}$ (lambda (x) $3x^4$))

Monovariant Flow Analysis: 0-CFA

(define ($\mathcal{D}$ f:($\lambda x\ 2x^3$) $\cup$ ($\lambda x\ 3x^4$))
...:($\lambda x\ 6x^2$))

($\mathcal{D}$ (lambda (x) $2x^3$)):($\lambda x\ 6x^2$)

($\mathcal{D}$ (lambda (x) $3x^4$))

Monovariant Flow Analysis: 0-CFA

(define ($\mathcal{D}$ f:$(\lambda x\ 2x^3) \cup (\lambda x\ 3x^4)$)
...:$(\lambda x\ 6x^2) \cup (\lambda x\ 12x^3)$)

($\mathcal{D}$ (lambda (x) $2x^3$)):$(\lambda x\ 6x^2)$

($\mathcal{D}$ (lambda (x) $3x^4$))

Monovariant Flow Analysis: 0-CFA

(define ($\mathcal{D}$ f:$(\lambda x\ 2x^3) \cup (\lambda x\ 3x^4)$)
...:$(\lambda x\ 6x^2) \cup (\lambda x\ 12x^3)$)

($\mathcal{D}$ (lambda (x) $2x^3$)):$(\lambda x\ 6x^2)$

($\mathcal{D}$ (lambda (x) $3x^4$)):$(\lambda x\ 12x^3)$

Monovariant Flow Analysis: 0-CFA

`(define (`$\mathcal{D}$ `f:`$(\lambda x\ 2x^3) \cup (\lambda x\ 3x^4)$`)`
` ...:`$(\lambda x\ 6x^2) \cup (\lambda x\ 12x^3)$`)`

`(`$\mathcal{D}$ `(lambda (x)` $2x^3$`)):`$(\lambda x\ 6x^2) \cup (\lambda x\ 12x^3)$

`(`$\mathcal{D}$ `(lambda (x)` $3x^4$`)):`$(\lambda x\ 6x^2) \cup (\lambda x\ 12x^3)$

Monovariant Flow Analysis: 0-CFA

`(define (`$\mathcal{D}$ `f:`$(\lambda x\ 2x^3) \cup (\lambda x\ 3x^4)$`)`
` ...:`$(\lambda x\ 6x^2) \cup (\lambda x\ 12x^3)$`)`

`(`$\mathcal{D}$ `(lambda (x)` $2x^3$`)):`$(\lambda x\ 6x^2) \cup (\lambda x\ 12x^3)$

`(`$\mathcal{D}$ `(lambda (x)` $3x^4$`)):`$(\lambda x\ 6x^2) \cup (\lambda x\ 12x^3)$

Monovariant Flow Analysis: 0-CFA

```
(define (𝒟 f)
 ...)
```

Monovariant Flow Analysis: 0-CFA

```
(define (𝒟 f)
 ...)
```

(𝒟 (𝒟 (lambda (x) e^{2x})))

Monovariant Flow Analysis: 0-CFA

```
(define (𝒟 f:(λx e^{2x}))
 ...)

(𝒟 (𝒟 (lambda (x) e^{2x})))
```

Monovariant Flow Analysis: 0-CFA

(define ($\mathcal{D}$ f:$(\lambda x\, e^{2x})$)
 ...:$(\lambda x\, 2e^{2x})$)

($\mathcal{D}$ ($\mathcal{D}$ (lambda (x) e^{2x})))

Monovariant Flow Analysis: 0-CFA

```
(define (𝒟 f:(λx e^{2x}))
 ...:(λx 2e^{2x}))
```

(define ($\mathcal{D}$ f:$(\lambda x\, e^{2x})$)
 ...:$(\lambda x\, 2e^{2x})$)

($\mathcal{D}$ ($\mathcal{D}$ (lambda (x) e^{2x}))):$(\lambda x\, 2e^{2x})$)

Monovariant Flow Analysis: 0-CFA

(define ($\mathcal{D}$ f:$(\lambda x\, e^{2x})\cup(\lambda x\, 2e^{2x})$)
 ...:$(\lambda x\, 2e^{2x})$)

($\mathcal{D}$ ($\mathcal{D}$ (lambda (x) e^{2x})):$(\lambda x\, 2e^{2x})$)

Monovariant Flow Analysis: 0-CFA

(define ($\mathcal{D}$ f:$(\lambda x\ e^{2x}) \cup (\lambda x\ 2e^{2x})$)
...:$(\lambda x\ 2e^{2x}) \cup (\lambda x\ 4e^{2x})$)

($\mathcal{D}$ ($\mathcal{D}$ (lambda (x) e^{2x})):$(\lambda x\ 2e^{2x})$)

Monovariant Flow Analysis: 0-CFA

(define ($\mathcal{D}$ f:$(\lambda x\, e^{2x}) \cup (\lambda x\, 2e^{2x})$)
 ...:$(\lambda x\, 2e^{2x}) \cup (\lambda x\, 4e^{2x})$)

($\mathcal{D}$ ($\mathcal{D}$ (lambda (x) e^{2x})):$(\lambda x\, 2e^{2x}) \cup (\lambda x\, 4e^{2x})$)

Monovariant Flow Analysis: 0-CFA

(define ($\mathcal{D}$ f:$(\lambda x\ e^{2x}) \cup (\lambda x\ 2e^{2x}) \cup (\lambda x\ 4e^{2x})$)
...:$(\lambda x\ 2e^{2x}) \cup (\lambda x\ 4e^{2x})$)

($\mathcal{D}$ ($\mathcal{D}$ (lambda (x) e^{2x}))):$(\lambda x\ 2e^{2x}) \cup (\lambda x\ 4e^{2x})$)

Monovariant Flow Analysis: 0-CFA

`(define (`$\mathcal{D}$ `f:`$(\lambda x\ e^{2x}) \cup (\lambda x\ 2e^{2x}) \cup (\lambda x\ 4e^{2x}) \cup \ldots$`)`
`...:`$(\lambda x\ 2e^{2x}) \cup (\lambda x\ 4e^{2x}) \cup \ldots$`)`

`(`$\mathcal{D}$ `(`$\mathcal{D}$ `(lambda (x)` e^{2x}`)):`$(\lambda x\ 2e^{2x}) \cup (\lambda x\ 4e^{2x}) \cup \ldots$`)`

Polyvariant Flow Analysis: *k*-CFA

with Bounded Context Sensitivity

```
(define (𝒟 f) ...)
```

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

Polyvariant Flow Analysis: *k*-CFA
with Bounded Context Sensitivity

```
(define (𝒟 f) ...)
```

```
(define (g ...) ... (𝒟 (lambda (x) 2x³)) ...)
```

(define (g ...) ... ($\mathcal{D}$ (lambda (x) $2x^3$)) ...)

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

Polyvariant Flow Analysis: *k*-CFA

with Bounded Context Sensitivity

```
(define (𝒟_g f) ...)
```

```
(define (g ...) ... (𝒟 (lambda (x) 2x³)) ...)
```

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

Polyvariant Flow Analysis: k-CFA

with Bounded Context Sensitivity

(define ($\mathcal{D}_{\texttt{g}}$ f:($\lambda x\ 2x^3$)) ...)

(define (g ...) ... ($\mathcal{D}$ (lambda (x) $2x^3$)) ...)

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

Polyvariant Flow Analysis: k-CFA

with Bounded Context Sensitivity

(define ($\mathcal{D}_{\texttt{g}}$ f:($\lambda x\ 2x^3$)) ...:($\lambda x\ 6x^2$))

(define (g ...) ... ($\mathcal{D}$ (lambda (x) $2x^3$)) ...)

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

Polyvariant Flow Analysis: k-CFA

with Bounded Context Sensitivity

(define ($\mathcal{D}_{\texttt{g}}$ f:($\lambda x\ 2x^3$)) ...:($\lambda x\ 6x^2$))

(define (g ...) ... ($\mathcal{D}$ (lambda (x) $2x^3$)):($\lambda x\ 6x^2$) ...)

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

Polyvariant Flow Analysis: k-CFA

with Bounded Context Sensitivity

(define ($\mathcal{D}_{\texttt{g}}$ f:($\lambda x\ 2x^3$)) ...:($\lambda x\ 6x^2$))

(define (g ...) ... ($\mathcal{D}$ (lambda (x) $2x^3$)):($\lambda x\ 6x^2$) ...)

(define (h ...) ... ($\mathcal{D}$ (lambda (x) $3x^4$)) ...)

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

Polyvariant Flow Analysis: k-CFA

with Bounded Context Sensitivity

```
(define (𝒟g f:(λx 2x³)) ...:(λx 6x²))

(define (𝒟h f) ...)

(define (g ...) ... (𝒟 (lambda (x) 2x³)):(λx 6x²) ...)

(define (h ...) ... (𝒟 (lambda (x) 3x⁴)) ...)
```

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

Polyvariant Flow Analysis: *k*-CFA

with Bounded Context Sensitivity

(define ($\mathcal{D}_{\texttt{g}}$ f:($\lambda x\ 2x^3$)) ...:($\lambda x\ 6x^2$))

(define ($\mathcal{D}_{\texttt{h}}$ f:($\lambda x\ 3x^4$)) ...)

(define (g ...) ... ($\mathcal{D}$ (lambda (x) $2x^3$)):($\lambda x\ 6x^2$) ...)

(define (h ...) ... ($\mathcal{D}$ (lambda (x) $3x^4$)) ...)

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

Polyvariant Flow Analysis: k-CFA

with Bounded Context Sensitivity

(define ($\mathcal{D}_{\texttt{g}}$ f:($\lambda x\ 2x^3$)) ...:($\lambda x\ 6x^2$))

(define ($\mathcal{D}_{\texttt{h}}$ f:($\lambda x\ 3x^4$)) ...:($\lambda x\ 12x^3$))

(define (g ...) ... ($\mathcal{D}$ (lambda (x) $2x^3$)):($\lambda x\ 6x^2$) ...)

(define (h ...) ... ($\mathcal{D}$ (lambda (x) $3x^4$)) ...)

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

Polyvariant Flow Analysis: *k*-CFA

with Bounded Context Sensitivity

(define ($\mathcal{D}_{\texttt{g}}$ f:($\lambda x\ 2x^3$)) ...:($\lambda x\ 6x^2$))

(define ($\mathcal{D}_{\texttt{h}}$ f:($\lambda x\ 3x^4$)) ...:($\lambda x\ 12x^3$))

(define (g ...) ... ($\mathcal{D}$ (lambda (x) $2x^3$)):($\lambda x\ 6x^2$) ...)

(define (h ...) ... ($\mathcal{D}$ (lambda (x) $3x^4$)):($\lambda x\ 12x^3$) ...)

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

Polyvariant Flow Analysis: *k*-CFA

with Bounded Context Sensitivity

```
(define ((compose n f) x)
 (if (zero? n) x ((compose (- n 1) f) (f x))))
```

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

Polyvariant Flow Analysis: *k*-CFA

with Bounded Context Sensitivity

```
(define ((compose n f) x)
 (if (zero? n) x ((compose (- n 1) f) (f x))))

((compose k 𝒟) g)
```

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

Polyvariant Flow Analysis: *k*-CFA

with Bounded Context Sensitivity

```
(define ((compose n f) x)
 (if (zero? n) x ((compose (- n 1) f) (f x))))
```

((compose k $\mathcal{D}$) g)

(define ($\mathcal{D}_{\texttt{compose}}$ f:g) ...)

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

Polyvariant Flow Analysis: *k*-CFA

with Bounded Context Sensitivity

```
(define ((compose n f) x)
 (if (zero? n) x ((compose (- n 1) f) (f x))))
```

((compose k $\mathcal{D}$) g)

(define ($\mathcal{D}_{\texttt{compose}}$ f:g) ...)

(define ($\mathcal{D}_{\texttt{compose:compose}}$ f:g') ...)

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

Polyvariant Flow Analysis: *k*-CFA

with Bounded Context Sensitivity

```
(define ((compose n f) x)
 (if (zero? n) x ((compose (- n 1) f) (f x))))
```

((compose k $\mathcal{D}$) g)

(define ($\mathcal{D}_{\texttt{compose}}$ f:g) ...)

(define ($\mathcal{D}_{\texttt{compose:compose}}$ f:g') ...)

(define ($\mathcal{D}_{\texttt{compose:compose:compose}}$ f:g'') ...)

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

Polyvariant Flow Analysis: *k*-CFA

with Bounded Context Sensitivity

```
(define ((compose n f) x)
 (if (zero? n) x ((compose (- n 1) f) (f x))))
```

((compose k $\mathcal{D}$) g)

(define ($\mathcal{D}_{\texttt{compose}}$ f:g) ...)

(define ($\mathcal{D}_{\texttt{compose:compose}}$ f:g') ...)

(define ($\mathcal{D}_{\texttt{compose:compose:compose}}$ f:g'') ...)

⋮

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

Polyvariant Flow Analysis

with Unbounded Context Sensitivity

$$\mathcal{E} : e \times \sigma \to v$$

Polyvariant Flow Analysis
with Unbounded Context Sensitivity

$$\mathcal{E} : e \times \sigma \rightarrow v$$

$$v ::= \texttt{\#t} \mid \texttt{\#f} \mid \texttt{()} \mid \mathbb{R} \mid (v_1, v_2) \mid \langle \sigma, e \rangle$$

$$\sigma ::= \{x_1 \mapsto v_1, \ldots\}$$

Polyvariant Flow Analysis

with Unbounded Context Sensitivity

$$\mathcal{E} : e \times \sigma \to v$$

$$v ::= \texttt{\#t} \mid \texttt{\#f} \mid \texttt{()} \mid \mathbb{R} \mid (v_1, v_2) \mid \langle \sigma, e \rangle$$

$$\sigma ::= \{x_1 \mapsto v_1, \ldots\}$$

$$\overline{v} ::= \texttt{\#t} \mid \texttt{\#f} \mid \texttt{()} \mid \mathbb{R} \mid (\overline{v_1}, \overline{v_2}) \mid \langle \overline{\sigma}, e \rangle \mid \overline{\mathbb{R}}$$

$$\overline{\sigma} ::= \{x_1 \mapsto \overline{v_1}, \ldots\}$$

Polyvariant Flow Analysis
with Unbounded Context Sensitivity

$$\mathcal{E} : e \times \sigma \rightarrow v$$

$$v ::= \texttt{\#t} \mid \texttt{\#f} \mid \texttt{()} \mid \mathbb{R} \mid (v_1, v_2) \mid \langle \sigma, e \rangle$$

$$\sigma ::= \{x_1 \mapsto v_1, \ldots\}$$

$$\overline{\mathcal{E}} : e \times \overline{\sigma} \rightarrow \overline{v}$$

$$\overline{v} ::= \texttt{\#t} \mid \texttt{\#f} \mid \texttt{()} \mid \mathbb{R} \mid (\overline{v_1}, \overline{v_2}) \mid \langle \overline{\sigma}, e \rangle \mid \overline{\mathbb{R}}$$

$$\overline{\sigma} ::= \{x_1 \mapsto \overline{v_1}, \ldots\}$$

Polyvariant Flow Analysis

with Unbounded Context Sensitivity

$$\mathcal{E} : e \times \sigma \rightarrow v$$

$$v ::= \texttt{\#t} \mid \texttt{\#f} \mid \texttt{()} \mid \mathbb{R} \mid (v_1, v_2) \mid \langle \sigma, e \rangle$$

$$\sigma ::= \{x_1 \mapsto v_1, \ldots\}$$

$$\overline{\mathcal{E}} : e \times \overline{\sigma} \rightarrow \overline{v}$$

$$\overline{v} ::= \texttt{\#t} \mid \texttt{\#f} \mid \texttt{()} \mid \mathbb{R} \mid (\overline{v_1}, \overline{v_2}) \mid \langle \overline{\sigma}, e \rangle \mid \overline{\mathbb{R}}$$

$$\overline{\sigma} ::= \{x_1 \mapsto \overline{v_1}, \ldots\}$$

Memoize $\overline{\mathcal{E}}$ indexed (by suitable equivalence relations on) e and $\overline{\sigma}$.

Polyvariant Flow Analysis

with Unbounded Context Sensitivity

$$\mathcal{E} : e \times \sigma \rightarrow v$$

$$v ::= \texttt{\#t} \mid \texttt{\#f} \mid \texttt{()} \mid \mathbb{R} \mid (v_1, v_2) \mid \langle \sigma, e \rangle$$

$$\sigma ::= \{x_1 \mapsto v_1, \ldots\}$$

$$\overline{\mathcal{E}} : e \times \overline{\sigma} \rightarrow \overline{v}$$

$$\overline{v} ::= \texttt{\#t} \mid \texttt{\#f} \mid \texttt{()} \mid \mathbb{R} \mid (\overline{v_1}, \overline{v_2}) \mid \langle \overline{\sigma}, e \rangle \mid \overline{\mathbb{R}}$$

$$\overline{\sigma} ::= \{x_1 \mapsto \overline{v_1}, \ldots\}$$

Memoize $\overline{\mathcal{E}}$ indexed (by suitable equivalence relations on) e and $\overline{\sigma}$.

Not suitable for arbitrary (i.e., typical SCHEME, ML, HASKELL, etc.) programs.

Polyvariant Flow Analysis

with Unbounded Context Sensitivity

$$\mathcal{E} : e \times \sigma \to v$$

$$v ::= \texttt{\#t} \mid \texttt{\#f} \mid \texttt{()} \mid \mathbb{R} \mid (v_1, v_2) \mid \langle \sigma, e \rangle$$

$$\sigma ::= \{x_1 \mapsto v_1, \ldots\}$$

$$\overline{\mathcal{E}} : e \times \overline{\sigma} \to \overline{v}$$

$$\overline{v} ::= \texttt{\#t} \mid \texttt{\#f} \mid \texttt{()} \mid \mathbb{R} \mid (\overline{v_1}, \overline{v_2}) \mid \langle \overline{\sigma}, e \rangle \mid \overline{\mathbb{R}}$$

$$\overline{\sigma} ::= \{x_1 \mapsto \overline{v_1}, \ldots\}$$

Memoize $\overline{\mathcal{E}}$ indexed (by suitable equivalence relations on) e and $\overline{\sigma}$.
Not suitable for arbitrary (i.e., typical SCHEME, ML, HASKELL, etc.) programs.
*I*s suitable for FORTRAN-like programs.

Polyvariant Flow Analysis

with Unbounded Context Sensitivity

$$\mathcal{E} : e \times \sigma \rightarrow v$$

$$v ::= \texttt{\#t} \mid \texttt{\#f} \mid \texttt{()} \mid \mathbb{R} \mid (v_1, v_2) \mid \langle \sigma, e \rangle$$

$$\sigma ::= \{x_1 \mapsto v_1, \ldots\}$$

$$\overline{\mathcal{E}} : e \times \overline{\sigma} \rightarrow \overline{v}$$

$$\overline{v} ::= \texttt{\#t} \mid \texttt{\#f} \mid \texttt{()} \mid \mathbb{R} \mid (\overline{v_1}, \overline{v_2}) \mid \langle \overline{\sigma}, e \rangle \mid \overline{\mathbb{R}}$$

$$\overline{\sigma} ::= \{x_1 \mapsto \overline{v_1}, \ldots\}$$

Memoize $\overline{\mathcal{E}}$ indexed (by suitable equivalence relations on) e and $\overline{\sigma}$.
Not suitable for arbitrary (i.e., typical SCHEME, ML, HASKELL, etc.) programs.
Is suitable for FORTRAN-like programs.
Necessary for migrating reflective source-code transformation to compile time.

Polyvariant Flow Analysis

with Unbounded Context Sensitivity

$$\mathcal{E} : e \times \sigma \rightarrow v$$

$$v ::= \texttt{\#t} \mid \texttt{\#f} \mid \texttt{()} \mid \mathbb{R} \mid (v_1, v_2) \mid \langle \sigma, e \rangle$$

$$\sigma ::= \{x_1 \mapsto v_1, \ldots\}$$

$$\overline{\mathcal{E}} : e \times \overline{\sigma} \rightarrow \overline{v}$$

$$\overline{v} ::= \texttt{\#t} \mid \texttt{\#f} \mid \texttt{()} \mid \mathbb{R} \mid (\overline{v_1}, \overline{v_2}) \mid \langle \overline{\sigma}, e \rangle \mid \overline{\mathbb{R}}$$

$$\overline{\sigma} ::= \{x_1 \mapsto \overline{v_1}, \ldots\}$$

Memoize $\overline{\mathcal{E}}$ indexed (by suitable equivalence relations on) e and $\overline{\sigma}$.
Not suitable for arbitrary (i.e., typical SCHEME, ML, HASKELL, etc.) programs.
*I*s suitable for FORTRAN-like programs.
Necessary for migrating reflective source-code transformation to compile time.
Side benefit: union-free

Polyvariant Flow Analysis

with Unbounded Context Sensitivity

$$\mathcal{E} : e \times \sigma \to v$$

$$v ::= \texttt{\#t} \mid \texttt{\#f} \mid \texttt{()} \mid \mathbb{R} \mid (v_1, v_2) \mid \langle \sigma, e \rangle$$

$$\sigma ::= \{x_1 \mapsto v_1, \ldots\}$$

$$\overline{\mathcal{E}} : e \times \overline{\sigma} \to \overline{v}$$

$$\overline{v} ::= \texttt{\#t} \mid \texttt{\#f} \mid \texttt{()} \mid \mathbb{R} \mid (\overline{v_1}, \overline{v_2}) \mid \langle \overline{\sigma}, e \rangle \mid \overline{\mathbb{R}}$$

$$\overline{\sigma} ::= \{x_1 \mapsto \overline{v_1}, \ldots\}$$

Memoize $\overline{\mathcal{E}}$ indexed (by suitable equivalence relations on) e and $\overline{\sigma}$.
Not suitable for arbitrary (i.e., typical SCHEME, ML, HASKELL, etc.) programs.
Is suitable for FORTRAN-like programs.
Necessary for migrating reflective source-code transformation to compile time.
Side benefit: union-free
No tags, tag checking, tag dispatching, indirect calls

Polyvariant Flow Analysis

with Unbounded Context Sensitivity

$$\mathcal{E} : e \times \sigma \rightarrow v$$

$$v ::= \texttt{\#t} \mid \texttt{\#f} \mid \texttt{()} \mid \mathbb{R} \mid (v_1, v_2) \mid \langle \sigma, e \rangle$$

$$\sigma ::= \{x_1 \mapsto v_1, \ldots\}$$

$$\overline{\mathcal{E}} : e \times \overline{\sigma} \rightarrow \overline{v}$$

$$\overline{v} ::= \texttt{\#t} \mid \texttt{\#f} \mid \texttt{()} \mid \mathbb{R} \mid (\overline{v_1}, \overline{v_2}) \mid \langle \overline{\sigma}, e \rangle \mid \overline{\mathbb{R}}$$

$$\overline{\sigma} ::= \{x_1 \mapsto \overline{v_1}, \ldots\}$$

Memoize $\overline{\mathcal{E}}$ indexed (by suitable equivalence relations on) e and $\overline{\sigma}$.
Not suitable for arbitrary (i.e., typical SCHEME, ML, HASKELL, etc.) programs.
Is suitable for FORTRAN-like programs.
Necessary for migrating reflective source-code transformation to compile time.
Side benefits: union-free, no cyclic abstract values
No tags, tag checking, tag dispatching, indirect calls

Polyvariant Flow Analysis

with Unbounded Context Sensitivity

$$\mathcal{E} : e \times \sigma \to v$$

$$v ::= \texttt{\#t} \mid \texttt{\#f} \mid \texttt{()} \mid \mathbb{R} \mid (v_1, v_2) \mid \langle \sigma, e \rangle$$

$$\sigma ::= \{x_1 \mapsto v_1, \ldots\}$$

$$\overline{\mathcal{E}} : e \times \overline{\sigma} \to \overline{v}$$

$$\overline{v} ::= \texttt{\#t} \mid \texttt{\#f} \mid \texttt{()} \mid \mathbb{R} \mid (\overline{v_1}, \overline{v_2}) \mid \langle \overline{\sigma}, e \rangle \mid \overline{\mathbb{R}}$$

$$\overline{\sigma} ::= \{x_1 \mapsto \overline{v_1}, \ldots\}$$

Memoize $\overline{\mathcal{E}}$ indexed (by suitable equivalence relations on) e and $\overline{\sigma}$.
Not suitable for arbitrary (i.e., typical SCHEME, ML, HASKELL, etc.) programs.
Is suitable for FORTRAN-like programs.
Necessary for migrating reflective source-code transformation to compile time.
Side benefits: union-free, no cyclic abstract values
No tags, tag checking, tag dispatching, indirect calls
Allows complete unboxing: no allocation, reclamation, indirection

Game Theory

				B		
		b_1	$\ldots$	b_j	$\ldots$	b_n
	a_1					
	$\vdots$		$\ddots$	$\vdots$		
A	a_i		$\ldots$	$\mathrm{PAYOFF}(a_i, b_j)$	$\ldots$	
	$\vdots$			$\vdots$	$\ddots$	
	a_m					

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

Game Theory

$$
\begin{array}{cc|ccccc}
 & & & & B & & \\
 & & b_1 & \ldots & b_j & \ldots & b_n \\
\hline
 & a_1 & & & & & \\
 & \vdots & & \ddots & \vdots & & \\
A & a_i & & \ldots & \textsc{Payoff}(a_i, b_j) & \ldots & \\
 & \vdots & & & \vdots & \ddots & \\
 & a_m & & & & &
\end{array}
$$

$$\max_{a \in A} \min_{b \in B} \textsc{Payoff}(a, b)$$

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

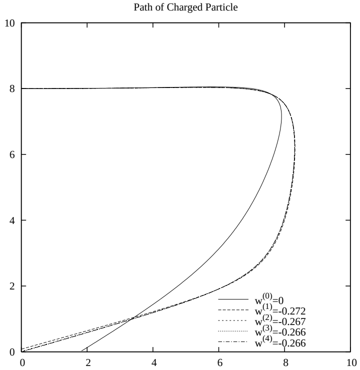
Game Theory

			$\mathbb{R}^n$	
		$\ldots$	$\mathbf{b}$	$\ldots$
	$\vdots$	$\ddots$	$\vdots$	
$\mathbb{R}^m$	$\mathbf{a}$	$\ldots$	$\textsc{Payoff}(\mathbf{a},\mathbf{b})$	$\ldots$
	$\vdots$		$\vdots$	$\ddots$

$$\max_{\mathbf{a}\in\mathbb{R}^m}\min_{\mathbf{b}\in\mathbb{R}^n}\textsc{Payoff}(\mathbf{a},\mathbf{b})$$

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

Cathode Ray Tubes



$$
\begin{aligned}
\text{potential: } p(\mathbf{x}; w) &= \|\mathbf{x} - (10, 10 - w)\|^{-1} + \|\mathbf{x} - (10, 0)\|^{-1} \\
\ddot{\mathbf{x}}(t) &= -\nabla_{\mathbf{x}}\, p(\mathbf{x})|_{\mathbf{x}=\mathbf{x}(t)} \\
\dot{\mathbf{x}}(t + \Delta t) &= \dot{\mathbf{x}}(t) + \Delta t\, \ddot{\mathbf{x}}(t) \\
\mathbf{x}(t + \Delta t) &= \mathbf{x}(t) + \Delta t\, \dot{\mathbf{x}}(t) \\
\text{When: } x_1(t + \Delta t) &\leq 0 \\
\text{let: } \Delta t_f &= -x_1(t)/\dot{x}_1(t) \\
t_f &= t + \Delta t_f \\
\mathbf{x}(t_f) &= \mathbf{x}(t) + \Delta t_f\, \dot{\mathbf{x}}(t) \\
\text{Error: } E(w) &= x_0(t_f)^2 \\
\text{Find: } & \operatorname*{argmin}_w E(w)
\end{aligned}
$$

Sprague, C. S. and George, R. H. (1939). *Cathode Ray Deflecting Electrode*. US Patent 2,161,437.

George, R. H. (1940). *Cathode Ray Tube*. US Patent 2,222,942.

Performance Comparison

		particle				saddle			
		FF	FR	RF	RR	FF	FR	RF	RR
VLAD	STALIN∇	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
FORTRAN	ADIFOR	1.52	🟦	🟦	🟦	2.07	🟦	🟦	🟦
	TAPENADE	3.40	🟥	🟩	🟥	2.56	🟥	🟩	🟥
C++	FADBAD++	65.69	🟩	🟩	🟩	22.44	🟩	🟩	🟩
ML	MLTON	53.89	88.88	16.08	28.06	40.39	51.21	1.86	2.67
	OCAML	160.50	340.35	147.91	263.66	107.71	156.33	6.75	13.51
	SML/NJ	106.21	182.45	105.04	185.15	84.38	106.01	3.55	6.31
HASKELL	GHC	165.22	🟥	🟥	🟥	121.18	🟥	🟥	🟥
SCHEME	BIGLOO	505.90	761.40	104.81	228.56	423.69	440.25	15.77	24.59
	CHICKEN	1120.37	2026.31	425.60	1872.85	889.58	1144.65	35.73	68.94
	GAMBIT	444.13	752.63	138.34	256.30	362.65	420.48	14.08	23.87
	IKARUS	192.07	312.28	61.79	114.87	158.88	205.97	6.75	11.40
	LARCENY	726.59	1108.18	144.55	270.14	571.81	613.65	19.14	29.77
	MIT SCHEME	1472.26	2500.00	309.66	591.36	1243.26	1428.57	51.36	79.10
	MZC	2073.26	3434.64	340.30	655.83	2436.26	1996.40	72.45	150.02
	MZSCHEME	2344.70	4076.16	409.95	843.68	2000.89	2332.43	80.78	134.00
	SCHEME->C	391.42	605.26	109.77	198.43	324.95	328.84	12.74	18.28
	SCMUTILS	3321.20	🟦	🟦	🟦	2800.71	🟦	🟦	🟦
	STALIN	208.10	366.08	51.84	91.86	166.96	212.93	7.68	11.40

- 🟩 not implemented but could implement
- 🟦 not implemented in existing tool
- 🟥 can't implement

Gradient-Based Optimization

```
(define (e i n)
 (if (zero? n)
     '()
     (cons (if (zero? i) 1.0 0.0)
           (e (- i 1) (- n 1)))))
```

Gradient-Based Optimization

```
(define (e i n)
 (if (zero? n)
     '()
     (cons (if (zero? i) 1.0 0.0)
           (e (- i 1) (- n 1)))))

(define ((gradient f) x)
 (let ((n (length x)))
  (map (lambda (i) (tangent ((j* f) (bundle x (e i n)))))
       (iota n))))
```

Gradient-Based Optimization

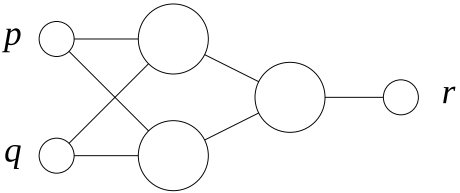
```
(define (e i n)
 (if (zero? n)
     '()
     (cons (if (zero? i) 1.0 0.0)
           (e (- i 1) (- n 1)))))

(define ((gradient f) x)
 (let ((n (length x)))
  (map (lambda (i) (tangent ((j* f) (bundle x (e i n)))))
       (iota n))))

(define (gradient-ascent f x0 n eta)
 (if (zero? n)
     (list x0 (f x0) ((gradient f) x0))
     (gradient-ascent f
                      (zip (lambda (xi gi) (+ xi (* eta gi)))
                           x0
                           ((gradient f) x0))
                      (- n 1)
                      eta)))
```

Gradient-Based Optimization

```
(define ((gradient f) x) (cdr ((cdr ((*j f) (*j x))) 1.0)))
```

```
(define (gradient-ascent f x0 n eta)
 (if (zero? n)
     (list x0 (f x0) ((gradient f) x0))
     (gradient-ascent f
                      (zip (lambda (xi gi) (+ xi (* eta gi)))
                           x0
                           ((gradient f) x0))
                      (- n 1)
                      eta)))
```

Neural Networks



p	q	r
0	0	0
0	1	1
1	0	1
1	1	0

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

Neural Networks in VLAD

```
(define ((sum-activities activities) bias ws)
 ((fold + bias) (zip * ws activities)))
```

Neural Networks in VLAD

```
(define ((sum-activities activities) bias ws)
 ((fold + bias) (zip * ws activities)))

(define (sum-layer activities ws-layer)
 (map (sum-activities activities) ws-layer))
```

Neural Networks in VLAD

```
(define ((sum-activities activities) bias ws)
 ((fold + bias) (zip * ws activities)))

(define (sum-layer activities ws-layer)
 (map (sum-activities activities) ws-layer))

(define (sigmoid x) (/ 1 (+ (exp (- 0 x)) 1)))
```

Neural Networks in VLAD

```
(define ((sum-activities activities) bias ws)
 ((fold + bias) (zip * ws activities)))

(define (sum-layer activities ws-layer)
 (map (sum-activities activities) ws-layer))

(define (sigmoid x) (/ 1 (+ (exp (- 0 x)) 1)))

(define ((forward-pass ws-layers) in)
 (if (null? ws-layers)
     in
     ((forward-pass (cdr ws-layers))
      (map sigmoid (sum-layer in (car ws-layers))))))
```

Neural Networks in VLAD

```
(define ((sum-activities activities) bias ws)
 ((fold + bias) (zip * ws activities)))

(define (sum-layer activities ws-layer)
 (map (sum-activities activities) ws-layer))

(define (sigmoid x) (/ 1 (+ (exp (- 0 x)) 1)))

(define ((forward-pass ws-layers) in)
 (if (null? ws-layers)
     in
     ((forward-pass (cdr ws-layers))
      (map sigmoid (sum-layer in (car ws-layers))))))

(define ((error-on-dataset dataset) ws-layers)
 ((fold + 0)
  (map (lambda ((list in target))
        (* 0.5 (magnitude-squared (v- ((forward-pass ws-layers) in) target))))
       dataset)))
```

Neural Networks in VLAD

```
(define ((sum-activities activities) bias ws)
 ((fold + bias) (zip * ws activities)))

(define (sum-layer activities ws-layer)
 (map (sum-activities activities) ws-layer))

(define (sigmoid x) (/ 1 (+ (exp (- 0 x)) 1)))

(define ((forward-pass ws-layers) in)
 (if (null? ws-layers)
     in
     ((forward-pass (cdr ws-layers))
      (map sigmoid (sum-layer in (car ws-layers))))))

(define ((error-on-dataset dataset) ws-layers)
 ((fold + 0)
  (map (lambda ((list in target))
        (* 0.5 (magnitude-squared (v- ((forward-pass ws-layers) in) target))))
       dataset)))

(gradient-descent (error-on-dataset '((((0 0) (0))
                                        ((0 1) (1))
                                        ((1 0) (1))
                                        ((1 1) (0))))
                  '(((0 -0.284227 1.16054) (0 0.617194 1.30467))
                    ((0 -0.084395 0.648461)))
                  1000.0
                  0.3)
```

Performance Comparison

		backprop		
		Fs	Fv	R
VLAD	STALIN∇	1.00	🟩	1.00
FORTRAN	ADIFOR	11.84	2.68	🟦
	TAPENADE	11.35	4.33	6.24
C	ADIC	16.33	3.93	🟦
C++	ADOL–C	12.34	3.89	35.53
	CPPAD	42.15	🟦	23.69
	FADBAD++	98.96	33.15	53.03
ML	MLTON	73.94	🟩	37.94
	OCAML	157.75	🟩	149.14
	SML/NJ	142.71	🟩	94.97
HASKELL	GHC	🟩	🟩	🟥
SCHEME	BIGLOO	577.45	🟩	306.60
	CHICKEN	1391.75	🟩	971.91
	GAMBIT	545.20	🟩	341.73
	IKARUS	216.42	🟩	147.49
	LARCENY	955.98	🟩	486.64
	MIT SCHEME	1900.04	🟩	1141.22
	MZC	2439.93	🟩	1571.52
	MZSCHEME	3477.86	🟩	1866.28
	SCHEME->C	484.24	🟩	233.75
	SCMUTILS	4544.48	🟦	🟦
	STALIN	832.68	🟩	367.84

- 🟩 not implemented but could implement
- 🟦 not implemented in existing tool
- 🟥 can't implement

Probabilistic Lambda Calculus

$P = \textbf{if } x_0 \textbf{ then } 0 \textbf{ else if } x_1 \textbf{ then } 1 \textbf{ else } 2$

Koller, D., McAllester, D. , and Pfeffer, A. (1997). *Effective Bayesian Inference for Stochastic Programs*. Proceedings of the 14th National Conference on Artificial Intelligence (AAAI), pp. 740–7.

Probabilistic Lambda Calculus

$P = \textbf{if } x_0 \textbf{ then } 0 \textbf{ else if } x_1 \textbf{ then } 1 \textbf{ else } 2$

$\Pr(x_0 \mapsto \textbf{true}) = p_0$ $\quad\quad\quad$ $\Pr(x_0 \mapsto \textbf{false}) = 1 - p_0$

$\Pr(x_1 \mapsto \textbf{true}) = p_1$ $\quad\quad\quad$ $\Pr(x_1 \mapsto \textbf{false}) = 1 - p_1$

Koller, D., McAllester, D. , and Pfeffer, A. (1997). *Effective Bayesian Inference for Stochastic Programs*. Proceedings of the 14th National Conference on Artificial Intelligence (AAAI), pp. 740–7.

Probabilistic Lambda Calculus

$P = \textbf{if } x_0 \textbf{ then } 0 \textbf{ else if } x_1 \textbf{ then } 1 \textbf{ else } 2$

$\Pr(x_0 \mapsto \textbf{true}) = p_0$ $\Pr(x_0 \mapsto \textbf{false}) = 1 - p_0$
$\Pr(x_1 \mapsto \textbf{true}) = p_1$ $\Pr(x_1 \mapsto \textbf{false}) = 1 - p_1$

$$\Pr(\mathcal{E}(P) = 0|p_0, p_1) = p_0$$
$$\Pr(\mathcal{E}(P) = 1|p_0, p_1) = (1 - p_0)p_1$$
$$\Pr(\mathcal{E}(P) = 2|p_0, p_1) = (1 - p_0)(1 - p_1)$$

Koller, D., McAllester, D. , and Pfeffer, A. (1997). *Effective Bayesian Inference for Stochastic Programs*. Proceedings of the 14th National Conference on Artificial Intelligence (AAAI), pp. 740–7.

Probabilistic Lambda Calculus

$P = \textbf{if } x_0 \textbf{ then } 0 \textbf{ else if } x_1 \textbf{ then } 1 \textbf{ else } 2$

$\Pr(x_0 \mapsto \textbf{true}) = p_0$ $\Pr(x_0 \mapsto \textbf{false}) = 1 - p_0$

$\Pr(x_1 \mapsto \textbf{true}) = p_1$ $\Pr(x_1 \mapsto \textbf{false}) = 1 - p_1$

$$\Pr(\mathcal{E}(P) = 0|p_0, p_1) = p_0$$
$$\Pr(\mathcal{E}(P) = 1|p_0, p_1) = (1 - p_0)p_1$$
$$\Pr(\mathcal{E}(P) = 2|p_0, p_1) = (1 - p_0)(1 - p_1)$$

$$\prod_{v \in \{0,1,2,2\}} \Pr(\mathcal{E}(P) = v|p_0, p_1) = p_0(1 - p_0)^3 p_1(1 - p_1)^2$$

Koller, D., McAllester, D. , and Pfeffer, A. (1997). *Effective Bayesian Inference for Stochastic Programs*. Proceedings of the 14th National Conference on Artificial Intelligence (AAAI), pp. 740–7.

Probabilistic Lambda Calculus

$P = \textbf{if } x_0 \textbf{ then } 0 \textbf{ else if } x_1 \textbf{ then } 1 \textbf{ else } 2$

$\Pr(x_0 \mapsto \textbf{true}) = p_0 \qquad \Pr(x_0 \mapsto \textbf{false}) = 1 - p_0$
$\Pr(x_1 \mapsto \textbf{true}) = p_1 \qquad \Pr(x_1 \mapsto \textbf{false}) = 1 - p_1$

$$\Pr(\mathcal{E}(P) = 0|p_0, p_1) = p_0$$
$$\Pr(\mathcal{E}(P) = 1|p_0, p_1) = (1 - p_0)p_1$$
$$\Pr(\mathcal{E}(P) = 2|p_0, p_1) = (1 - p_0)(1 - p_1)$$

$$\prod_{v \in \{0,1,2,2\}} \Pr(\mathcal{E}(P) = v|p_0, p_1) = p_0(1 - p_0)^3 p_1(1 - p_1)^2$$

$$\operatorname*{argmax}_{p_0, p_1} \prod_{v \in \{0,1,2,2\}} \Pr(\mathcal{E}(P) = v|p_0, p_1) = \left\langle \frac{1}{4}, \frac{1}{3} \right\rangle$$

Koller, D., McAllester, D. , and Pfeffer, A. (1997). *Effective Bayesian Inference for Stochastic Programs*. Proceedings of the 14th National Conference on Artificial Intelligence (AAAI), pp. 740–7.

Probabilistic Prolog

```
p(0).
p(X):-q(X).
q(1).
q(2).
```

Probabilistic Prolog

$\Pr(\texttt{p(0).}) = p_0$
$\Pr(\texttt{p(X):-q(X).}) = 1 - p_0$
$\Pr(\texttt{q(1).}) = p_1$
$\Pr(\texttt{q(2).}) = 1 - p_1$

Probabilistic Prolog

$\Pr(\texttt{p(0).}) = p_0$
$\Pr(\texttt{p(X):-q(X).}) = 1 - p_0$
$\Pr(\texttt{q(1).}) = p_1$
$\Pr(\texttt{q(2).}) = 1 - p_1$

$\Pr(\texttt{?-p(0).}) = p_0$
$\Pr(\texttt{?-p(1).}) = (1 - p_0)p_1$
$\Pr(\texttt{?-p(2).}) = (1 - p_0)(1 - p_1)$

Probabilistic Prolog

$$\Pr(\texttt{p(0).}) = p_0$$
$$\Pr(\texttt{p(X):-q(X).}) = 1 - p_0$$
$$\Pr(\texttt{q(1).}) = p_1$$
$$\Pr(\texttt{q(2).}) = 1 - p_1$$

$$\Pr(\texttt{?-p(0).}) = p_0$$
$$\Pr(\texttt{?-p(1).}) = (1 - p_0)p_1$$
$$\Pr(\texttt{?-p(2).}) = (1 - p_0)(1 - p_1)$$

$$\prod_{q \in \{\texttt{p(0)},\texttt{p(1)},\texttt{p(2)},\texttt{p(2)}\}} \Pr(\texttt{?-}q\texttt{.}) = p_0(1 - p_0)^3 p_1(1 - p_1)^2$$

Probabilistic Prolog

$\Pr(\texttt{p(0).}) = p_0$
$\Pr(\texttt{p(X):-q(X).}) = 1 - p_0$
$\Pr(\texttt{q(1).}) = p_1$
$\Pr(\texttt{q(2).}) = 1 - p_1$

$\Pr(\texttt{?-p(0).}) = p_0$
$\Pr(\texttt{?-p(1).}) = (1 - p_0)p_1$
$\Pr(\texttt{?-p(2).}) = (1 - p_0)(1 - p_1)$

$$\prod_{q \in \{\texttt{p(0)},\texttt{p(1)},\texttt{p(2)},\texttt{p(2)}\}} \Pr(\texttt{?-}q\texttt{.}) = p_0(1 - p_0)^3 p_1 (1 - p_1)^2$$

$$\operatorname*{argmax}_{p_0,p_1} \prod_{q \in \{\texttt{p(0)},\texttt{p(1)},\texttt{p(2)},\texttt{p(2)}\}} \Pr(\texttt{?-}q\texttt{.}) = \left\langle \frac{1}{4}, \frac{1}{3} \right\rangle$$

Probabilistic Lambda Calculus

```
(define (evaluate expression environment)
 (cond
  ((constant-expression? expression)
   (singleton-tagged-distribution
    (constant-expression-value expression)))
  ((variable-access-expression? expression)
   (lookup-value
    (variable-access-expression-variable expression) environment))
  ((lambda-expression? expression)
   (singleton-tagged-distribution
    (lambda (tagged-distribution)
     (evaluate
      (lambda-expression-body expression)
      (cons (make-binding (lambda-expression-variable expression)
                          tagged-distribution)
            environment)))))
  (else (let ((tagged-distribution
               (evaluate (application-argument expression)
                         environment)))
         (map-tagged-distribution
          (lambda (value) (value tagged-distribution))
          (evaluate (application-callee expression) environment))))))
```

Probabilistic Lambda Calculus

```
(define (evaluate expression environment)
 (cond
  ((constant-expression? expression)
   (singleton-tagged-distribution
    (constant-expression-value expression)))
  ((variable-access-expression? expression)
   (lookup-value
    (variable-access-expression-variable expression) environment))
  ((lambda-expression? expression)
   (singleton-tagged-distribution
    (lambda (tagged-distribution)
     (evaluate
      (lambda-expression-body expression)
      (cons (make-binding (lambda-expression-variable expression)
                          tagged-distribution)
            environment))))))
  (else (let ((tagged-distribution
               (evaluate (application-argument expression)
                         environment)))
         (map-tagged-distribution
          (lambda (value) (value tagged-distribution))
          (evaluate (application-callee expression) environment))))))
```

Probabilistic Lambda Calculus

```
(define (evaluate expression environment)
 (cond
  ((constant-expression? expression)
   (singleton-tagged-distribution
    (constant-expression-value expression)))
  ((variable-access-expression? expression)
   (lookup-value
    (variable-access-expression-variable expression) environment))
  ((lambda-expression? expression)
   (singleton-tagged-distribution
    (lambda (tagged-distribution)
     (evaluate
      (lambda-expression-body expression)
      (cons (make-binding (lambda-expression-variable expression)
                          tagged-distribution)
            environment)))))
  (else (let ((tagged-distribution
               (evaluate (application-argument expression)
                         environment)))
         (map-tagged-distribution
          (lambda (value) (value tagged-distribution))
          (evaluate (application-callee expression) environment))))))
```

Probabilistic Lambda Calculus

```
(define (evaluate expression environment)
 (cond
  ((constant-expression? expression)
   (singleton-tagged-distribution
    (constant-expression-value expression)))
  ((variable-access-expression? expression)
   (lookup-value
    (variable-access-expression-variable expression) environment))
  ((lambda-expression? expression)
   (singleton-tagged-distribution
    (lambda (tagged-distribution)
     (evaluate
      (lambda-expression-body expression)
      (cons (make-binding (lambda-expression-variable expression)
                          tagged-distribution)
            environment)))))
  (else (let ((tagged-distribution
               (evaluate (application-argument expression)
                         environment)))
         (map-tagged-distribution
          (lambda (value) (value tagged-distribution))
          (evaluate (application-callee expression) environment))))))
```

Probabilistic Lambda Calculus

```
(define (evaluate expression environment)
 (cond
  ((constant-expression? expression)
   (singleton-tagged-distribution
    (constant-expression-value expression)))
  ((variable-access-expression? expression)
   (lookup-value
    (variable-access-expression-variable expression) environment))
  ((lambda-expression? expression)
   (singleton-tagged-distribution
    (lambda (tagged-distribution)
     (evaluate
      (lambda-expression-body expression)
      (cons (make-binding (lambda-expression-variable expression)
                          tagged-distribution)
            environment)))))
  (else (let ((tagged-distribution
               (evaluate (application-argument expression)
                         environment)))
         (map-tagged-distribution
          (lambda (value) (value tagged-distribution))
          (evaluate (application-callee expression) environment))))))
```

Probabilistic Lambda Calculus

```
(define (evaluate expression environment)
 (cond
  ((constant-expression? expression)
   (singleton-tagged-distribution
    (constant-expression-value expression)))
  ((variable-access-expression? expression)
   (lookup-value
    (variable-access-expression-variable expression) environment))
  ((lambda-expression? expression)
   (singleton-tagged-distribution
    (lambda (tagged-distribution)
     (evaluate
      (lambda-expression-body expression)
      (cons (make-binding (lambda-expression-variable expression)
                          tagged-distribution)
            environment)))))
  (else (let ((tagged-distribution
               (evaluate (application-argument expression)
                         environment)))
         (map-tagged-distribution
          (lambda (value) (value tagged-distribution))
          (evaluate (application-callee expression) environment))))))
```

Probabilistic Lambda Calculus

```
(define (evaluate expression environment)
 (cond
  ((constant-expression? expression)
   (singleton-tagged-distribution
    (constant-expression-value expression)))
  ((variable-access-expression? expression)
   (lookup-value
    (variable-access-expression-variable expression) environment))
  ((lambda-expression? expression)
   (singleton-tagged-distribution
    (lambda (tagged-distribution)
     (evaluate
      (lambda-expression-body expression)
      (cons (make-binding (lambda-expression-variable expression)
                          tagged-distribution)
            environment)))))
  (else (let ((tagged-distribution
               (evaluate (application-argument expression)
                         environment)))
         (map-tagged-distribution
          (lambda (value) (value tagged-distribution))
          (evaluate (application-callee expression) environment))))))
```

Probabilistic Lambda Calculus

```
(define (evaluate expression environment)
 (cond
  ((constant-expression? expression)
   (singleton-tagged-distribution
    (constant-expression-value expression)))
  ((variable-access-expression? expression)
   (lookup-value
    (variable-access-expression-variable expression) environment))
  ((lambda-expression? expression)
   (singleton-tagged-distribution
    (lambda (tagged-distribution)
     (evaluate
      (lambda-expression-body expression)
      (cons (make-binding (lambda-expression-variable expression)
                          tagged-distribution)
            environment)))))
  (else (let ((tagged-distribution
               (evaluate (application-argument expression)
                         environment)))
         (map-tagged-distribution
          (lambda (value) (value tagged-distribution))
          (evaluate (application-callee expression) environment))))))
```

Probabilistic Lambda Calculus

```
(define (evaluate expression environment)
 (cond
  ((constant-expression? expression)
   (singleton-tagged-distribution
    (constant-expression-value expression)))
  ((variable-access-expression? expression)
   (lookup-value
    (variable-access-expression-variable expression) environment))
  ((lambda-expression? expression)
   (singleton-tagged-distribution
    (lambda (tagged-distribution)
     (evaluate
      (lambda-expression-body expression)
      (cons (make-binding (lambda-expression-variable expression)
                          tagged-distribution)
            environment)))))
  (else (let ((tagged-distribution
               (evaluate (application-argument expression)
                         environment)))
        (map-tagged-distribution
         (lambda (value) (value tagged-distribution))
         (evaluate (application-callee expression) environment))))))
```

Probabilistic Lambda Calculus

```
(define (evaluate expression environment)
 (cond
  ((constant-expression? expression)
   (singleton-tagged-distribution
    (constant-expression-value expression)))
  ((variable-access-expression? expression)
   (lookup-value
    (variable-access-expression-variable expression) environment))
  ((lambda-expression? expression)
   (singleton-tagged-distribution
    (lambda (tagged-distribution)
     (evaluate
      (lambda-expression-body expression)
      (cons (make-binding (lambda-expression-variable expression)
                          tagged-distribution)
            environment))))))
  (else (let ((tagged-distribution
               (evaluate (application-argument expression)
                         environment)))
         (map-tagged-distribution
          (lambda (value) (value tagged-distribution))
          (evaluate (application-callee expression) environment))))))
```

Probabilistic Lambda Calculus

```
(gradient-ascent
 (lambda (p)
  (let ((tagged-distribution
          (evaluate if x0 then 0 else if x1 then 1 else 2
                    (list Pr(x0 ↦ true) = p0   Pr(x0 ↦ false) = 1 − p0
                          Pr(x1 ↦ true) = p1   Pr(x1 ↦ false) = 1 − p1
                          ...))))
   (map-reduce
    *
    1.0
    (lambda (value)
     (likelihood value tagged-distribution))
    '(0 1 2 2))))
 '(0.5 0.5)
 1000.0
 0.1)
```

Probabilistic Lambda Calculus

```
(gradient-ascent
 (lambda (p)
  (let ((tagged-distribution
         (evaluate if x0 then 0 else if x1 then 1 else 2
                   (list Pr(x0 ↦ true) = p0   Pr(x0 ↦ false) = 1 − p0
                         Pr(x1 ↦ true) = p1   Pr(x1 ↦ false) = 1 − p1
                         ...))))
   (map-reduce
    *
    1.0
    (lambda (value)
     (likelihood value tagged-distribution))
    '(0 1 2 2))))
 '(0.5 0.5)
 1000.0
 0.1)
```

Probabilistic Lambda Calculus

```
(gradient-ascent
 (lambda (p)
  (let ((tagged-distribution
         (evaluate if x0 then 0 else if x1 then 1 else 2
                   (list Pr(x0 ↦ true) = p0   Pr(x0 ↦ false) = 1 − p0
                         Pr(x1 ↦ true) = p1   Pr(x1 ↦ false) = 1 − p1
                         ...))))
   (map-reduce
    *
    1.0
    (lambda (value)
     (likelihood value tagged-distribution))
    '(0 1 2 2))))
 '(0.5 0.5)
 1000.0
 0.1)
```

Probabilistic Lambda Calculus

```
(gradient-ascent
 (lambda (p)
  (let ((tagged-distribution
         (evaluate if x0 then 0 else if x1 then 1 else 2
                   (list Pr(x0 ↦ true) = p0   Pr(x0 ↦ false) = 1 − p0
                         Pr(x1 ↦ true) = p1   Pr(x1 ↦ false) = 1 − p1
                         ...)))))
   (map-reduce
    *
    1.0
    (lambda (value)
     (likelihood value tagged-distribution))
    '(0 1 2 2))))
 '(0.5 0.5)
 1000.0
 0.1)
```

Probabilistic Lambda Calculus

```
(gradient-ascent
 (lambda (p)
  (let ((tagged-distribution
         (evaluate if x0 then 0 else if x1 then 1 else 2
                   (list Pr(x0 ↦ true) = p0   Pr(x0 ↦ false) = 1 − p0
                         Pr(x1 ↦ true) = p1   Pr(x1 ↦ false) = 1 − p1
                         ...)))))
   (map-reduce
    *
    1.0
    (lambda (value)
     (likelihood value tagged-distribution))
    '(0 1 2 2))))
 '(0.5 0.5)
 1000.0
 0.1)
```

Probabilistic Lambda Calculus

```
(gradient-ascent
 (lambda (p)
  (let ((tagged-distribution
         (evaluate if x0 then 0 else if x1 then 1 else 2
                   (list Pr(x0 ↦ true) = p0   Pr(x0 ↦ false) = 1 − p0
                         Pr(x1 ↦ true) = p1   Pr(x1 ↦ false) = 1 − p1
                         ...))))
   (map-reduce
    *
    1.0
    (lambda (value)
     (likelihood value tagged-distribution))
    '(0 1 2 2))))
 '(0.5 0.5)
 1000.0
 0.1)
```

Probabilistic Lambda Calculus

```
(gradient-ascent
 (lambda (p)
  (let ((tagged-distribution
         (evaluate if x0 then 0 else if x1 then 1 else 2
                   (list Pr(x0 ↦ true) = p0   Pr(x0 ↦ false) = 1 − p0
                         Pr(x1 ↦ true) = p1   Pr(x1 ↦ false) = 1 − p1
                         ...)))))
   (map-reduce
    *
    1.0
    (lambda (value)
     (likelihood value tagged-distribution))
    '(0 1 2 2))))
 '(0.5 0.5)
 1000.0
 0.1)
```

Probabilistic Lambda Calculus

```
(gradient-ascent
 (lambda (p)
  (let ((tagged-distribution
         (evaluate if x0 then 0 else if x1 then 1 else 2
                   (list Pr(x0 ↦ true) = p0   Pr(x0 ↦ false) = 1 − p0
                         Pr(x1 ↦ true) = p1   Pr(x1 ↦ false) = 1 − p1
                         ...))))
   (map-reduce
    *
    1.0
    (lambda (value)
     (likelihood value tagged-distribution))
    '(0 1 2 2))))
 '(0.5 0.5)
 1000.0
 0.1)
```

Probabilistic Lambda Calculus

```
(gradient-ascent
 (lambda (p)
  (let ((tagged-distribution
         (evaluate if x0 then 0 else if x1 then 1 else 2
                   (list Pr(x0 ↦ true) = p0   Pr(x0 ↦ false) = 1 − p0
                         Pr(x1 ↦ true) = p1   Pr(x1 ↦ false) = 1 − p1
                         ...))))
   (map-reduce
    *
    1.0
    (lambda (value)
     (likelihood value tagged-distribution))
    '(0 1 2 2))))
 '(0.5 0.5)
 1000.0
 0.1)
```

Probabilistic Lambda Calculus

```
(gradient-ascent
 (lambda (p)
  (let ((tagged-distribution
         (evaluate if x0 then 0 else if x1 then 1 else 2
                   (list Pr(x0 ↦ true) = p0   Pr(x0 ↦ false) = 1 − p0
                         Pr(x1 ↦ true) = p1   Pr(x1 ↦ false) = 1 − p1
                         ...))))
   (map-reduce
    *
    1.0
    (lambda (value)
     (likelihood value tagged-distribution))
    '(0 1 2 2))))
 '(0.5 0.5)
 1000.0
 0.1)
```

Probabilistic Prolog

```
(define (proof-distribution term clauses)
 (let ((offset ...))
  (map-reduce
   append
   '()
   (lambda (clause)
    (let ((clause (alpha-rename clause offset)))
     (let loop ((p (clause-p clause))
                (substitution (unify term (clause-term clause)))
                (terms (clause-terms clause)))
      (if (boolean? substitution)
          '()
          (if (null? terms)
              (list (make-double p substitution))
              (map-reduce
               append
               '()
               (lambda (double)
                (loop (* p (double-p double))
                      (append substitution (double-substitution double))
                      (rest terms)))
               (proof-distribution
                (apply-substitution substitution (first terms)) clauses)))))))
   clauses)))
```

Probabilistic Prolog

```
(define (proof-distribution term clauses)
 (let ((offset ...))
  (map-reduce
   append
   '()
   (lambda (clause)
    (let ((clause (alpha-rename clause offset)))
     (let loop ((p (clause-p clause))
                (substitution (unify term (clause-term clause)))
                (terms (clause-terms clause)))
      (if (boolean? substitution)
          '()
          (if (null? terms)
              (list (make-double p substitution))
              (map-reduce
               append
               '()
               (lambda (double)
                (loop (* p (double-p double))
                      (append substitution (double-substitution double))
                      (rest terms)))
               (proof-distribution
                (apply-substitution substitution (first terms)) clauses)))))))
   clauses)))
```

Probabilistic Prolog

```
(define (proof-distribution term clauses)
 (let ((offset ...))
  (map-reduce
   append
   '()
   (lambda (clause)
    (let ((clause (alpha-rename clause offset)))
     (let loop ((p (clause-p clause))
                (substitution (unify term (clause-term clause)))
                (terms (clause-terms clause)))
      (if (boolean? substitution)
          '()
          (if (null? terms)
              (list (make-double p substitution))
              (map-reduce
               append
               '()
               (lambda (double)
                (loop (* p (double-p double))
                      (append substitution (double-substitution double))
                      (rest terms)))
               (proof-distribution
                (apply-substitution substitution (first terms)) clauses))))))))
   clauses)))
```

Probabilistic Prolog

```
(define (proof-distribution term clauses)
 (let ((offset ...))
  (map-reduce
   append
   '()
   (lambda (clause)
    (let ((clause (alpha-rename clause offset)))
     (let loop ((p (clause-p clause))
                (substitution (unify term (clause-term clause)))
                (terms (clause-terms clause)))
      (if (boolean? substitution)
          '()
          (if (null? terms)
              (list (make-double p substitution))
              (map-reduce
               append
               '()
               (lambda (double)
                (loop (* p (double-p double))
                      (append substitution (double-substitution double))
                      (rest terms)))
               (proof-distribution
                (apply-substitution substitution (first terms)) clauses)))))))
   clauses)))
```

Probabilistic Prolog

```
(define (proof-distribution term clauses)
 (let ((offset ...))
  (map-reduce
   append
   '()
   (lambda (clause)
    (let ((clause (alpha-rename clause offset)))
     (let loop ((p (clause-p clause))
                (substitution (unify term (clause-term clause)))
                (terms (clause-terms clause)))
      (if (boolean? substitution)
          '()
          (if (null? terms)
              (list (make-double p substitution))
              (map-reduce
               append
               '()
               (lambda (double)
                (loop (* p (double-p double))
                      (append substitution (double-substitution double))
                      (rest terms)))
               (proof-distribution
                (apply-substitution substitution (first terms)) clauses)))))))
   clauses)))
```

Probabilistic Prolog

```
(define (proof-distribution term clauses)
 (let ((offset ...))
  (map-reduce
   append
   '()
   (lambda (clause)
    (let ((clause (alpha-rename clause offset)))
     (let loop ((p (clause-p clause))
                (substitution (unify term (clause-term clause)))
                (terms (clause-terms clause)))
      (if (boolean? substitution)
          '()
          (if (null? terms)
              (list (make-double p substitution))
              (map-reduce
               append
               '()
               (lambda (double)
                (loop (* p (double-p double))
                      (append substitution (double-substitution double))
                      (rest terms)))
               (proof-distribution
                (apply-substitution substitution (first terms)) clauses)))))))
   clauses)))
```

Probabilistic Prolog

```
(define (proof-distribution term clauses)
 (let ((offset ...))
  (map-reduce
   append
   '()
   (lambda (clause)
    (let ((clause (alpha-rename clause offset)))
     (let loop ((p (clause-p clause))
                (substitution (unify term (clause-term clause)))
                (terms (clause-terms clause)))
      (if (boolean? substitution)
          '()
          (if (null? terms)
              (list (make-double p substitution))
              (map-reduce
               append
               '()
               (lambda (double)
                (loop (* p (double-p double))
                      (append substitution (double-substitution double))
                      (rest terms)))
               (proof-distribution
                (apply-substitution substitution (first terms)) clauses))))))
   clauses)))
```

Probabilistic Prolog

```
(define (proof-distribution term clauses)
 (let ((offset ...))
  (map-reduce
   append
   '()
   (lambda (clause)
    (let ((clause (alpha-rename clause offset)))
     (let loop ((p (clause-p clause))
                (substitution (unify term (clause-term clause)))
                (terms (clause-terms clause)))
      (if (boolean? substitution)
          '()
          (if (null? terms)
              (list (make-double p substitution))
              (map-reduce
               append
               '()
               (lambda (double)
                (loop (* p (double-p double))
                      (append substitution (double-substitution double))
                      (rest terms)))
               (proof-distribution
                (apply-substitution substitution (first terms)) clauses)))))))
   clauses)))
```

Probabilistic Prolog

```
(define (proof-distribution term clauses)
 (let ((offset ...))
  (map-reduce
   append
   '()
   (lambda (clause)
    (let ((clause (alpha-rename clause offset)))
     (let loop ((p (clause-p clause))
                (substitution (unify term (clause-term clause)))
                (terms (clause-terms clause)))
      (if (boolean? substitution)
          '()
          (if (null? terms)
              (list (make-double p substitution))
              (map-reduce
               append
               '()
               (lambda (double)
                (loop (* p (double-p double))
                      (append substitution (double-substitution double))
                      (rest terms)))
               (proof-distribution
                (apply-substitution substitution (first terms)) clauses)))))))
   clauses)))
```

Probabilistic Prolog

```
(define (proof-distribution term clauses)
 (let ((offset ...))
  (map-reduce
   append
   '()
   (lambda (clause)
    (let ((clause (alpha-rename clause offset)))
     (let loop ((p (clause-p clause))
                (substitution (unify term (clause-term clause)))
                (terms (clause-terms clause)))
      (if (boolean? substitution)
          '()
          (if (null? terms)
              (list (make-double p substitution))
              (map-reduce
               append
               '()
               (lambda (double)
                (loop (* p (double-p double))
                      (append substitution (double-substitution double))
                      (rest terms)))
               (proof-distribution
                (apply-substitution substitution (first terms)) clauses)))))))
   clauses)))
```

Probabilistic Prolog

```
(define (proof-distribution term clauses)
 (let ((offset ...))
  (map-reduce
   append
   '()
   (lambda (clause)
    (let ((clause (alpha-rename clause offset)))
     (let loop ((p (clause-p clause))
                (substitution (unify term (clause-term clause)))
                (terms (clause-terms clause)))
      (if (boolean? substitution)
          '()
          (if (null? terms)
              (list (make-double p substitution))
              (map-reduce
               append
               '()
               (lambda (double)
                (loop (* p (double-p double))
                      (append substitution (double-substitution double))
                      (rest terms)))
               (proof-distribution
                (apply-substitution substitution (first terms)) clauses)))))))
   clauses)))
```

Probabilistic Prolog

```
(gradient-ascent
 (lambda (p)
  (let ((clauses (list Pr(p(0).) = p0
                       Pr(p(X):-q(X).) = 1 - p0
                       Pr(q(1).) = p1
                       Pr(q(2).) = 1 - p1)))
   (map-reduce
    *
    1.0
    (lambda (query)
     (likelihood (proof-distribution query clauses)))
    '(p(0) p(1) p(2) p(2)))))
 '(0.5 0.5)
 1000.0
 0.1)
```

Probabilistic Prolog

```
(gradient-ascent
 (lambda (p)
  (let ((clauses (list Pr(p(0).) = p0
                       Pr(p(X):-q(X).) = 1 - p0
                       Pr(q(1).) = p1
                       Pr(q(2).) = 1 - p1)))
   (map-reduce
    *
    1.0
    (lambda (query)
     (likelihood (proof-distribution query clauses)))
    '(p(0) p(1) p(2) p(2)))))
 '(0.5 0.5)
 1000.0
 0.1)
```

Probabilistic Prolog

```
(gradient-ascent
 (lambda (p)
  (let ((clauses (list Pr(p(0).) = p0
                       Pr(p(X):-q(X).) = 1 - p0
                       Pr(q(1).) = p1
                       Pr(q(2).) = 1 - p1)))
   (map-reduce
    *
    1.0
    (lambda (query)
     (likelihood (proof-distribution query clauses)))
    '(p(0) p(1) p(2) p(2)))))
 '(0.5 0.5)
 1000.0
 0.1)
```

Probabilistic Prolog

```
(gradient-ascent
 (lambda (p)
  (let ((clauses (list Pr(p(0).) = p0
                       Pr(p(X):-q(X).) = 1 - p0
                       Pr(q(1).) = p1
                       Pr(q(2).) = 1 - p1)))
   (map-reduce
    *
    1.0
    (lambda (query)
     (likelihood (proof-distribution query clauses)))
    '(p(0) p(1) p(2) p(2)))))
 '(0.5 0.5)
 1000.0
 0.1)
```

Probabilistic Prolog

```
(gradient-ascent
 (lambda (p)
  (let ((clauses (list Pr(p(0).) = p0
                       Pr(p(X):-q(X).) = 1 - p0
                       Pr(q(1).) = p1
                       Pr(q(2).) = 1 - p1)))
   (map-reduce
    *
    1.0
    (lambda (query)
     (likelihood (proof-distribution query clauses)))
    '(p(0) p(1) p(2) p(2)))))
 '(0.5 0.5)
 1000.0
 0.1)
```

Probabilistic Prolog

```
(gradient-ascent
 (lambda (p)
  (let ((clauses (list Pr(p(0).) = p0
                       Pr(p(X):-q(X).) = 1 − p0
                       Pr(q(1).) = p1
                       Pr(q(2).) = 1 − p1)))
   (map-reduce
    *
    1.0
    (lambda (query)
     (likelihood (proof-distribution query clauses)))
    '(p(0) p(1) p(2) p(2)))))
 '(0.5 0.5)
 1000.0
 0.1)
```

Probabilistic Prolog

```
(gradient-ascent
 (lambda (p)
  (let ((clauses (list Pr(p(0).) = p0
                       Pr(p(X):-q(X).) = 1 - p0
                       Pr(q(1).) = p1
                       Pr(q(2).) = 1 - p1)))
   (map-reduce
    *
    1.0
    (lambda (query)
     (likelihood (proof-distribution query clauses)))
    '(p(0) p(1) p(2) p(2)))))
 '(0.5 0.5)
 1000.0
 0.1)
```

Generated Code

```
static void f2679(double a_f2679_0,double a_f2679_1,double a_f2679_2,double a_f2679_3){
  int t272381=((a_f2679_2==0.)?0:1);
  double t272406;
  double t272405;
  double t272404;
  double t272403;
  double t272402;
  if((t272381==0)){
    double t272480=(1.-a_f2679_0);
    double t272572=(1.-a_f2679_1);
    double t273043=(a_f2679_0+0.);
    double t274185=(t272480*a_f2679_1);
    double t274426=(t274185+0.);
    double t275653=(t272480*t272572);
    double t275894=(t275653+0.);
    double t277121=(t272480*t272572);
    double t277362=(t277121+0.);
    double t277431=(t277362*1.);
    double t277436=(t275894*t277431);
    double t277441=(t274426*t277436);
    double t277446=(t273043*t277441);
    ...
    double t1777107=(t1774696+t1715394);
    double t1777194=(0.-t1745420);
    double t1778533=(t1777194+t1419700);
    t272406=a_f2679_0;
    t272405=a_f2679_1;
    t272404=t277446;
    t272403=t1778533;
    t272402=t1777107;}
  else {...}
  r_f2679_0=t272406;
  r_f2679_1=t272405;
  r_f2679_2=t272404;
  r_f2679_3=t272403;
  r_f2679_4=t272402;}
```

Performance Comparison

		probabilistic-lambda-calculus		probabilistic-prolog	
		F	R	F	R
VLAD	STALIN∇	1.00	1.00	1.00	1.00
ML	MLTON	106.45	124.95	789.41	483.47
	OCAML	215.73	538.68	1207.13	1534.61
	SML/NJ	197.75	272.45	2448.02	1471.94
HASKELL	GHC	■ (green)	■ (red)	■ (green)	■ (red)
SCHEME	BIGLOO	832.92	1048.11	14422.16	8286.06
	CHICKEN	2305.98	3283.00	66948.70	37792.84
	GAMBIT	879.88	1153.86	24316.03	13649.81
	IKARUS	437.46	531.10	8242.92	4845.86
	LARCENY	1651.01	1673.22	25589.62	14833.53
	MIT SCHEME	3491.10	4130.19	85819.57	48335.38
	MZC	5289.17	5929.14	154206.95	83480.27
	MZSCHEME	6235.78	7134.71	166129.12	91630.70
	SCHEME->C	682.15	794.31	10530.66	5980.27
	SCMUTILS	6456.99	■ (blue)	80100.23	■ (blue)
	STALIN	1240.73	1137.41	22511.79	10986.43

- ■ (green) not implemented but could implement, including FORTRAN, C, and C++
- ■ (blue) not implemented in existing tool
- ■ (red) can't implement

Where You Can Read About This Work

Where You Can Read About This Work

- AD 2004(a,b,c,d), 2008(a,b)
- POPL 2005, 2006, 2008, 2009
- PLDI 2006, 2008, 2009
- ICFP 2007(a,b), 2008

Where You Can Read About This Work

- AD 20[illegible](a,b,c,d), 2008(a,b)
- POPL 200[illegible], 2006, 2008, 2009
- PLDI 2006, 20[illegible], 2009
- ICFP 2007(a,b), 20[illegible]8

Evaluation: C

It is not clear to me how this approach is better than automatic differentiation frameworks using C++ templates and the C++ type inference algorithm.

The paper seems correct, and I agree that it is hard to implement automatic differentiation both conveniently and efficiently without the proper programming language support (i.e., efficiently implemented generics with a minimal amount of type inference).

It's not clear to me that an efficient C++ templates implementation doesn't give the benefits listed here: it is "callee derives" (mostly) and efficiently implemented (using inlining). (The extent that it's not "callee derives" is that you need to put "template<class X>" in front of every function and declare all your variables as being of type "X" instead of "double".)

I was somewhat shocked that the only C++ implementation you compared to was apparently one that uses virtual function dispatch (FADBAD++). I Googled "automatic differentiation C++ template" and found two papers that do what I expected you to compare to:

Dan Piponi: Automatic Differentiation, C++ Templates and Photogrammetry, "to be published in the Journal of Graphics Tools", Sep. 2004.

and

M.E. Jerrell: Function Minimization and Automatic Differentiation Using C++, OOPSLA, 1989.

```
                         Evaluation:  C. Weak paper, but it will not be an
                                       embarrassment to have it in POPL.
                         Confidence:  Z. I am an informed outsider and tried
                                       my best to understand the paper.

                             ===== Summary =====

Shows how to optimize a functional language with a built-in automatic
differentiation operator.

                         ===== Detailed Comments =====

The results look useful, but I wonder whether POPL is the right place to
present them. Yes, the development involves functional programming. But it
also involves a lot of concepts from scientific computing that may be
unfamiliar to many and that are explained only minimally or not at all.
```

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.

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.

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