AD for Probabilistic Programming

Jeffrey Mark Siskind
qobi@purdue.edu

School of Electrical and Computer Engineering
Purdue University

Probabilistic Programming
Universal Languages, Systems and Applicationss
NIPS
13 December 2008

Joint work with Barak A. Pearlmutter.

The Essence

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

The Essence

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

The Essence

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

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 (`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) $2x^3$)

(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

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

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

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

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

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

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

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

The Essence

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

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

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

The Essence

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

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

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

The Essence

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

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

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

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

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

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

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

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

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

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`)`	$\Longrightarrow$	`(`$\mathcal{D}$ $\langle\{f \mapsto \lambda x\, 2x^3\}, \lambda x\ \sin f(x)\rangle$`)`
	$\Longrightarrow$	$\langle\{f \mapsto \lambda x\, 2x^3, f' \mapsto \lambda x\, 6x^2\},$ $\lambda x\, f'(x)\cos f(x)\rangle$
`(map-closure` f $\langle\{x_1 \mapsto v_1, \ldots\}, e\rangle$`)`	$\Longrightarrow$	$\langle\{x_1 \mapsto f(v_1), \ldots\}, e\rangle$

The Essence

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

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

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

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

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

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

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

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

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

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

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

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

The Essence

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

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

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

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

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

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

The Essence

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

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

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

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

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

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))
```

```
(𝒟 (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))

(𝒟 (𝒟 f))

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

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

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

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
```

```
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

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
```

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

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)                            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
```

```
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
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 $\overline{\mathbb{R}^{\acute{n}}}$

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

$$\begin{aligned}
\texttt{bundle} &: \mathbb{R}^n \times \overline{\mathbb{R}^n} \rightarrow (\mathbb{R}^n \rhd \overline{\mathbb{R}^n}) \\
\texttt{primal} &: (\mathbb{R}^n \rhd \overline{\mathbb{R}^n}) \rightarrow \mathbb{R}^n \\
\texttt{tangent} &: (\mathbb{R}^n \rhd \overline{\mathbb{R}^n}) \rightarrow \overline{\mathbb{R}^n}
\end{aligned}$$

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

Our API for Functional Forward AD

$$
\begin{aligned}
\texttt{bundle} &: \mathbb{R}^n \times \overline{\mathbb{R}^{\acute{n}}} \rightarrow (\mathbb{R}^n \rhd \overline{\mathbb{R}^{\acute{n}}}) \\
\texttt{primal} &: (\mathbb{R}^n \rhd \overline{\mathbb{R}^{\acute{n}}}) \rightarrow \mathbb{R}^n \\
\texttt{tangent} &: (\mathbb{R}^n \rhd \overline{\mathbb{R}^{\acute{n}}}) \rightarrow \overline{\mathbb{R}^{\acute{n}}} \\
\texttt{j*} &: {\color{red}(\mathbb{R}^n \rightarrow \mathbb{R}^m)} \rightarrow ((\mathbb{R}^n \rhd \overline{\mathbb{R}^{\acute{n}}}) \rightarrow (\mathbb{R}^m \rhd \overline{\mathbb{R}^{\acute{m}}}))
\end{aligned}
$$

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

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

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

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

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

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*

Our API for Functional Forward AD

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

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

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

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

Our API for Functional Forward AD

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

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*

Generalize to arbitrary types

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

Our API for Functional Forward AD

$$\texttt{bundle} : \tau \times \overrightharpoon{\tau} \rightarrow (\tau \rhd \overrightharpoon{\tau})$$

$$\texttt{primal} : (\tau \rhd \overrightharpoon{\tau}) \rightarrow \tau$$

$$\texttt{tangent} : (\tau \rhd \overrightharpoon{\tau}) \rightarrow \overrightharpoon{\tau}$$

$$\texttt{j*} : (\tau_1 \rightarrow \tau_2) \rightarrow ((\tau_1 \rhd \overrightharpoon{\tau_1}) \rightarrow (\tau_2 \rhd \overrightharpoon{\tau_2}))$$

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

`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 $\tau \rhd \overrightharpoon{\tau}$ as $\overrightarrow{\tau}$

Our API for Functional Forward AD

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

Differential geometry bundles points $\mathbb{R}^n$ in a manifold with tangent vectors $\overline{\mathbb{R}^n}$
$\texttt{j}\star$ 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 $\overrightarrow{\tau}$

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

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

```
(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 $\tau \triangleright \overline{\tau}$ as $\overrightharpoon{\tau}$
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)
```

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

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

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

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

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

```
(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 $\overrightarrow{\mathcal{J}}$

Convenient

Our API for Functional Forward AD

$$\texttt{bundle} : \tau \times \overrightarrow{\tau'} \rightarrow \overrightarrow{\tau}$$

$$\texttt{primal} : \overrightarrow{\tau} \rightarrow \tau$$

$$\texttt{tangent} : \overrightarrow{\tau} \rightarrow \overrightarrow{\tau'}$$

$$\texttt{j*} : (\tau_1 \rightarrow \tau_2) \rightarrow (\overrightarrow{\tau_1} \rightarrow \overrightarrow{\tau_2})$$

```
(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 $\overrightarrow{\mathbb{R}^{n}}'$
`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 $\overrightarrow{\mathcal{J}}$
What is `(j* j*)`?
Convenient

Our API for Functional Forward AD

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

```
(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 $\overrightarrow{\mathbb{R}^n}'$

`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 $\overrightarrow{\mathcal{J}}$

What is `(j* 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))
(𝒟 (lambda (x) ... (𝒟 (lambda (y) ...) ...) ...) ...)
```

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 $\overrightarrow{\mathcal{J}}$
What is `(j* j*)`?
Convenient and fast

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

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

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

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

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

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

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

$\textsc{NeutronFlux}\ r \quad \triangleq \quad$ *classified*

Modularity

$\nabla\ f\ \mathbf{x}$	$\triangleq$	$\frac{\partial f(\mathbf{x})}{\partial x_1}, \ldots, \frac{\partial f(\mathbf{x})}{\partial x_n}$
$\textsc{GradientDescent}\ f\ \mathbf{x}_0$	$\triangleq$	$\ldots \mathbf{x}_{i+1} := \ldots \nabla\ f\ \mathbf{x}_i \ldots$
$\text{argmin}\ f$	$\triangleq$	$\ldots \textsc{GradientDescent}\ f\ \mathbf{x}_0 \ldots$
$\textsc{NeutronFlux}\ r$	$\triangleq$	*classified*
$\textsc{Deviation}\ r$	$\triangleq$	$((\textsc{NeutronFlux}\ r) - \textsc{NeutronFlux}_{\text{critical}})^2$

Modularity

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

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

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

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

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

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

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

Breaking Modularity

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

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

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

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

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

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

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

Breaking Modularity

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

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

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

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

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

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

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

Breaking Modularity

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

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

Breaking Modularity

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

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

Breaking Modularity

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

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

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

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

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

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

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

Breaking Modularity

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Breaking Modularity

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Breaking Modularity

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

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

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

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

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

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

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

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

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

Breaking Modularity

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

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

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

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

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

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

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

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

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

Breaking Modularity

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

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

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

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

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

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

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

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

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

Breaking Modularity

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

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

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

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

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

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

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

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

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

Breaking Modularity

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Breaking Modularity

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

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

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

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

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

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

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

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

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

Breaking Modularity

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

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

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

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

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

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

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

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

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

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

Breaking Modularity

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

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

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

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

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

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

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

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

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

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

Breaking Modularity

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

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

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

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

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

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

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

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

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

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

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

Breaking Modularity

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

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

Breaking Modularity

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

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

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

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

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

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

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

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

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

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

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

Breaking Modularity

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

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

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

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

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

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

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

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

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

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

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

Breaking Modularity

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

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

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

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

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

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

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

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

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

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

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

Breaking Modularity

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

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

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

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

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

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

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

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

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

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

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

Breaking Modularity

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

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

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

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

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

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

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

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

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

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

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

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

Breaking Modularity

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

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

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

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

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

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

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

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

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

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

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

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

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

Restoring Modularity

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

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

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

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

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

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

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

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

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

Restoring Modularity

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

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

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

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

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

$\textsc{NeutronFlux}\ \mathbf{r} \quad \triangleq$ *classified*

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

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

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

Restoring Modularity

$\nabla\, f\ \mathbf{x}$	$\triangleq$	$\ldots(\overleftarrow{\mathcal{J}}\ f)\ \mathbf{x}\ldots$
$\mathcal{H}\, f\ \mathbf{x}$	$\triangleq$	
$\textsc{GradientDescent}\ f\ \mathbf{x}_0$	$\triangleq$	$\ldots\mathbf{x}_{i+1} := \ldots\nabla\, f\ \mathbf{x}_i\ldots$
$\textsc{NewtonsMethod}\ f\ \mathbf{x}_0$	$\triangleq$	$\ldots\mathbf{x}_{i+1} := \ldots\nabla\, f\ \mathbf{x}_i\ldots\mathcal{H}\, f\ \mathbf{x}_i\ldots$
$\operatorname{argmin}\ f$	$\triangleq$	$\ldots\textsc{GradientDescent}\ f\ \mathbf{x}_0\ldots$
$\textsc{NeutronFlux}\ \mathbf{r}$	$\triangleq$	*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

$$\nabla\ f\ \mathbf{x} \triangleq \ldots(\overleftarrow{\mathcal{J}}\ f)\ \mathbf{x}\ldots$$

$$\mathcal{H}\ f\ \mathbf{x} \triangleq \ldots(\overrightarrow{\mathcal{J}}\ (\overleftarrow{\mathcal{J}}\ f))\ldots\mathbf{x}\ldots$$

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

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

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

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

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

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

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

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
    ($\mathcal{D}$ ($\mathcal{D}$ `f`))
  - nesting
    ($\mathcal{D}$ `(lambda (...) ...` ($\mathcal{D}$ `(lambda (...) ...)) ...))`
- Fast
  - $\mathcal{D}$ implemented by reflective transformation of environments and code associated with closures
  - 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 ($\mathcal{D}$ f:($\lambda x\ 2x^3$))
 ...:($\lambda x\ 6x^2$))

($\mathcal{D}$ (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 (𝒟 f:(λx 2x³) ∪ (λx 3x⁴))
 ...:(λx 6x²) ∪ (λx 12x³))
```

(𝒟 (lambda (x) $2x^3$)):$(\lambda x\ 6x^2) \cup (\lambda x\ 12x^3)$

(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}))
 ...)
```

$(\mathcal{D}\ f{:}(\lambda x\ e^{2x}))$

$(\mathcal{D}\ (\mathcal{D}\ (\texttt{lambda}\ (\texttt{x})\ e^{2x})))$

Monovariant Flow Analysis: 0-CFA

```
(define (𝒟 f:(λx e^{2x}))
 ...:(λx 2e^{2x}))
```

$\texttt{(define (}\mathcal{D}\texttt{ f:}(\lambda x\, e^{2x})\texttt{)}$
$\texttt{ ...:}(\lambda x\, 2e^{2x})\texttt{)}$

$\texttt{(}\mathcal{D}\texttt{ (}\mathcal{D}\texttt{ (lambda (x) }e^{2x}\texttt{)))}$

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 (𝒟 f:(λx e^{2x}) ∪ (λx 2e^{2x}))
 ...:(λx 2e^{2x}))

(𝒟 (𝒟 (lambda (x) e^{2x})):(λx 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 ($\mathcal{D}$ f) ...)

(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 ($\mathcal{D}_{\mathtt{g}}$ f) ...)

(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 (`$\mathcal{D}_\mathtt{g}$` 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 ($\mathcal{D}_{\mathtt{g}}$ 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 ($\mathcal{D}_g$ 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 ($\mathcal{D}_{\texttt{g}}$ f:$(\lambda x\ 2x^3)$) ...:$(\lambda x\ 6x^2)$)

(define ($\mathcal{D}_{\texttt{h}}$ f) ...)

(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 ($\mathcal{D}_{\tt g}$ f:$(\lambda x\ 2x^3)$) ...:$(\lambda x\ 6x^2)$)

(define ($\mathcal{D}_{\tt h}$ 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''`) ...)`

$\vdots$

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

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

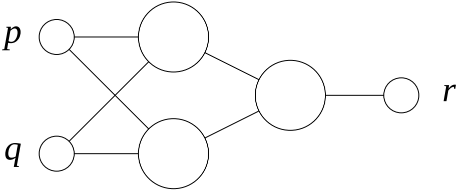
```
(define (e i n)
 (if (zero? n)
     '()
     (cons (if (zero? i) 1.0 0.0)
           (e (- i 1) (- n 1)))))

(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

	forward scalar	forward vector	reverse
STALIN∇	1.39		1.00
FADBAD++	103.65	35.38	52.40
ADOL–C	12.80	4.07	32.55
CPPAD	44.10		22.20
ADIC	17.50	4.13	
ADIFOR	12.38	2.79	
TAPENADE	11.86	4.54	5.80

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

$P = \textbf{if } x_0 \textbf{ then } 0 \textbf{ else if } x_1 \textbf{ then } 1 \textbf{ else } 2$

$\Pr(x_0 \mapsto \textbf{true}) = p_0$ $\Pr(x_0 \mapsto \textbf{false}) = 1 - p_0$
$\Pr(x_1 \mapsto \textbf{true}) = p_1$ $\Pr(x_1 \mapsto \textbf{false}) = 1 - p_1$

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

$P = \textbf{if } x_0 \textbf{ then } 0 \textbf{ else if } x_1 \textbf{ then } 1 \textbf{ else } 2$

$\Pr(x_0 \mapsto \textbf{true}) = p_0 \qquad \Pr(x_0 \mapsto \textbf{false}) = 1 - p_0$
$\Pr(x_1 \mapsto \textbf{true}) = p_1 \qquad \Pr(x_1 \mapsto \textbf{false}) = 1 - p_1$

$\Pr(\mathcal{E}(P) = 0 | p_0, p_1) = p_0$
$\Pr(\mathcal{E}(P) = 1 | p_0, p_1) = (1 - p_0)p_1$
$\Pr(\mathcal{E}(P) = 2 | p_0, p_1) = (1 - p_0)(1 - p_1)$

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

$P = \textbf{if } x_0 \textbf{ then } 0 \textbf{ else if } x_1 \textbf{ then } 1 \textbf{ else } 2$

$\Pr(x_0 \mapsto \textbf{true}) = p_0$ $\quad\quad$ $\Pr(x_0 \mapsto \textbf{false}) = 1 - p_0$
$\Pr(x_1 \mapsto \textbf{true}) = p_1$ $\quad\quad$ $\Pr(x_1 \mapsto \textbf{false}) = 1 - p_1$

$\Pr(\mathcal{E}(P) = 0 | p_0, p_1) = p_0$
$\Pr(\mathcal{E}(P) = 1 | p_0, p_1) = (1 - p_0)p_1$
$\Pr(\mathcal{E}(P) = 2 | p_0, p_1) = (1 - p_0)(1 - p_1)$

$$\prod_{v \in \{0,1,2,2\}} \Pr(\mathcal{E}(P) = v | p_0, p_1) = p_0(1 - p_0)^3 p_1 (1 - p_1)^2$$

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

$P =$ **if** x_0 **then** 0 **else if** x_1 **then** 1 **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** x_0 **then** 0 **else** **if** x_1 **then** 1 **else** 2
                   (list $\Pr(x_0 \mapsto \textbf{true}) = p_0$ $\Pr(x_0 \mapsto \textbf{false}) = 1 - p_0$
                         $\Pr(x_1 \mapsto \textbf{true}) = p_1$ $\Pr(x_1 \mapsto \textbf{false}) = 1 - p_1$
                         ...)))))
   (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 x_0 then 0 else if x_1 then 1 else 2
                   (list Pr(x_0 ↦ true) = p_0   Pr(x_0 ↦ false) = 1 − p_0
                         Pr(x_1 ↦ true) = p_1   Pr(x_1 ↦ false) = 1 − p_1
                         ...)))))
   (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).) = p_0
                       Pr(p(X):-q(X).) = 1 - p_0
                       Pr(q(1).) = p_1
                       Pr(q(2).) = 1 - p_1)))
   (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 $\Pr(\texttt{p(0).}) = p_0$
                       $\Pr(\texttt{p(X):-q(X).}) = 1 - p_0$
                       $\Pr(\texttt{q(1).}) = p_1$
                       $\Pr(\texttt{q(2).}) = 1 - p_1$)))
   (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	
	forward	reverse	forward	reverse
STALIN∇	1.00	1.00	1.00	1.00
IKARUS	499.13	419.37	10,384.61	4,347.82
SCHEME->C	934.34	660.69	11,394.23	5,838.50
BIGLOO	1,367.10	967.60	14,531.25	7,701.86
GAMBIT	1,155.64	1,035.55	24,831.73	12,931.67
LARCENY	2,294.07	1,412.75	24,471.15	12,906.83
STALIN	1,313.00	1,925.84	22,524.03	14,633.54
CHICKEN	2,168.67	2,659.16	53,320.31	28,434.78
SCMUTILS	6,448.24	3,449.89	85,697.11	43,416.14
MZC	5,227.44	5,325.36	144,765.62	118,192.54
MZSCHEME	8,667.32	6,370.29	157,965.74	124,975.15

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