Automatic differentiation

Compositional backpropagation and other methods

Bas van der Linden

Radboud University

21 January 2025

Papers

1. "Automatic differentiation in machine learning: a survey"
by Atılım Günes Baydin, Barak A. Pearlmutter, Alexey Andreyevich Radul and Jeffrey Mark Siskind (2017)

Papers

1. "Automatic differentiation in machine learning: a survey"
   by Atılım Günes Baydin, Barak A. Pearlmutter, Alexey Andreyevich Radul and Jeffrey Mark Siskind (2017)
2. "Backpropagation in the simply typed lambda-calculus with linear negation"
   by Aloïs Brunel, Damiano Mazza, and Michele Pagani (2020)

History of automatic differentiation

Computing derivative in computer programs.

1. Forward mode described in (one of the) first CS PhD dissertations in 1964.

History of automatic differentiation

Computing derivative in computer programs.

1. Forward mode described in (one of the) first CS PhD dissertations in 1964.
2. Origin of reverse mode is not entirely clear, but most likely a Finnish master thesis from 1970.

History of automatic differentiation

Computing derivative in computer programs.

1. Forward mode described in (one of the) first CS PhD dissertations in 1964.
2. Origin of reverse mode is not entirely clear, but most likely a Finnish master thesis from 1970.
3. Many usecases:
    1. Scientific computing
    2. Machine learning, although only (relatively) recently has general automatic differentiation been applied to it

What it isn't: Numerical differentiation

1. Finite difference methods
2. Easy to implement,

What it isn't: Numerical differentiation

1. Finite difference methods
2. Easy to implement,
   Definition of derivative: $f'(x) = \lim_{h \to 0} \frac{f(x+h)-f(x)}{h}$
   Using some small value dx, we can approximate the derivative: $Df(x, dx) = \frac{f(x+dx)-f(x)}{dx}$

What it isn't: Numerical differentiation

1. Finite difference methods
2. Easy to implement, $f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$
3. Inherently imprecise due to rounding and floating point truncation.
   We're adding a really small number to a (fairly) large number, and subtracting numbers that are almost the same.

What it isn't: Numerical differentiation

1. Finite difference methods
2. Easy to implement, $f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$
3. Inherently imprecise due to rounding and floating point truncation.
4. There are better methods that improve rounding errors, but they increase complexity and still suffer from truncation

What it isn't: Symbolic differentiation

1. Manipulating expressions using known rules
   Chain rule: $(f(g(x)))' = f'(g(x)) \cdot g'(x)$
   Product rule: $(f \cdot g)' = f' \cdot g + f \cdot g'$

What it isn't: Symbolic differentiation

1. Manipulating expressions using known rules
   Chain rule: $(f(g(x)))' = f'(g(x)) \cdot g'(x)$
   Product rule: $(f \cdot g)' = f' \cdot g + f \cdot g'$
2. CAS engines: Mathematica, Maxima, Maple

What it isn't: Symbolic differentiation

1. Manipulating expressions using known rules
   Chain rule: $(f(g(x)))' = f'(g(x)) \cdot g'(x)$
   Product rule: $(f \cdot g)' = f' \cdot g + f \cdot g'$
2. CAS engines: Mathematica, Maxima, Maple
3. Less efficient for runtime calculations, as expressions can grow exponentially

Automatic differentiation

1. Still based on chain rule, but we don't care about symbolic expression.

Automatic differentiation

1. Still based on chain rule, but we don't care about symbolic expression.
2. Based on principle that any computation is composition of elementary functions with known derivative

Automatic differentiation

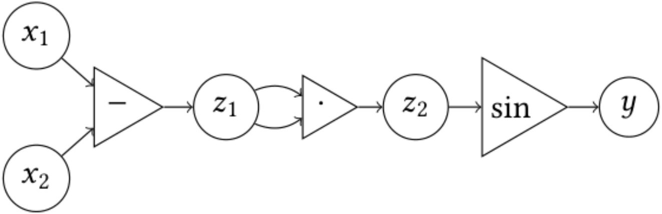
1. Still based on chain rule, but we don't care about symbolic expression.
2. Based on principle that any computation is composition of elementary functions with known derivative
3. Also allows to differentiate algorithms beyond closed-form expressions: using branching, loops etc.

Computational graph

We can express this composition of basic functions as a computational graph.

Computational graph

We can express this composition of basic functions as a computational graph.
$y = sin((x_1 - x_2) \cdot (x_1 - x_2))$ can be represented as

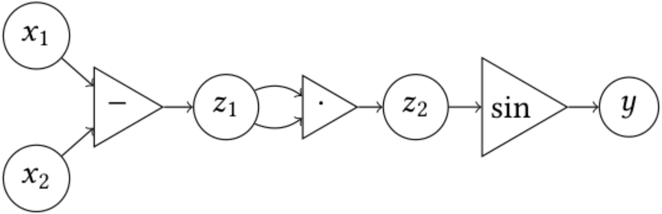


let z_1 = x_1 - x_2 **in let** z_2 = z_1 · z_1 **in** sin z_2

Triangles are the elementary functions, although I will leave them out later.

Computational graph

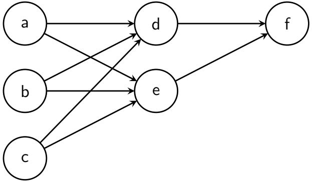
We can express this composition of basic functions as a computational graph.
$y = sin((x_1 - x_2) \cdot (x_1 - x_2))$ can be represented as




```
let z₁ = x₁ - x₂ in let z₂ = z₁ · z₁ in sin z₂
```

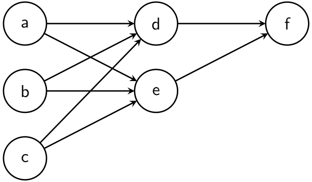
Triangles are the elementary functions, although I will leave them out later.
This construction allows node sharing, which is important for performance, as we only have to calculate things once.

Forward mode



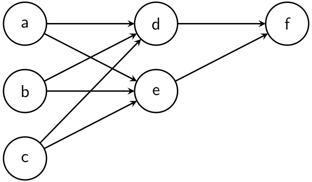
Let's say we want to compute $\frac{df}{da}$ with inputs i_1, i_2, i_3.
For any node x, by ϕ_x we denote the function of its inputs (instead of the triangles before).

Forward mode



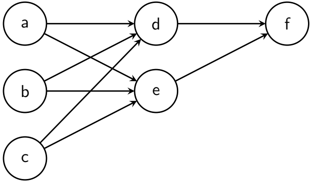
Let's say we want to compute $\frac{df}{da}$ with inputs i_1, i_2, i_3.
For any node x, by ϕ_x we denote the function of its inputs.
Idea: for each node x $\in$ {a,b,c,d,e,f} compute two values: x and x'.

Forward mode



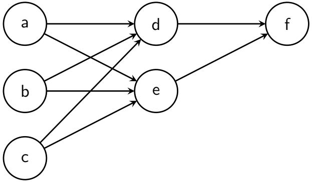
Let's say we want to compute $\frac{df}{da}$ with inputs i_1, i_2, i_3.
For any node x, by ϕ_x we denote the function of its inputs.
Idea: for each node x $\in$ {a,b,c,d,e,f} compute two values:
x and x'.
The value of that node.

Forward mode



Let's say we want to compute $\frac{df}{da}$ with inputs i_1, i_2, i_3.
For any node x, by ϕ_x we denote the function of its inputs.
Idea: for each node x $\in$ {a,b,c,d,e,f} compute two values: x and x'.
The derivative of that node to a,
so $x' = \frac{dx}{da}$

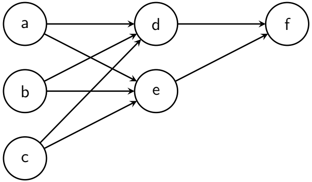
Forward mode



Let's say we want to compute $\frac{df}{da}$ with inputs i_1, i_2, i_3.
For any node x, by ϕ_x we denote the function of its inputs.
Idea: for each node x $\in$ {a,b,c,d,e,f} compute two values: x and x', where $x' = \frac{dx}{da}$.
We start by initializing the starting values of the nodes without inputs (so a,b,c).

$$a = i_1$$
$$b = i_2$$
$$c = i_3$$

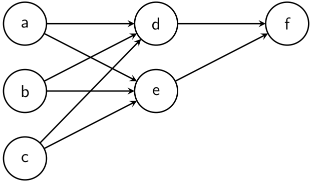
Forward mode



Let's say we want to compute $\frac{df}{da}$ with inputs i_1, i_2, i_3.
For any node x, by ϕ_x we denote the function of its inputs.
Idea: for each node x $\in$ {a,b,c,d,e,f} compute two values: x and x', where $x' = \frac{dx}{da}$.
We set a' to 1 because that's the variable to which we want the derivative. The derivatives of the other nodes are set to 0

$a = i_1$	$a' = 1$
$b = i_2$	$b' = 0$
$c = i_3$	$c' = 0$

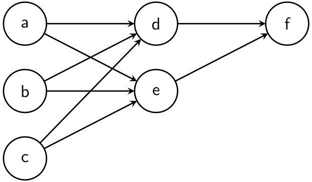
Forward mode



Let's say we want to compute $\frac{df}{da}$ with inputs i_1, i_2, i_3.
For any node x, by ϕ_x we denote the function of its inputs.
Idea: for each node x $\in$ {a,b,c,d,e,f} compute two values: x and x', where $x' = \frac{dx}{da}$.
For the further nodes we can compute the values directly and the derivatives using the chain rule.

$a = i_1$	$a' = 1$
$b = i_2$	$b' = 0$
$c = i_3$	$c' = 0$
$d = \phi_d(a, b, c)$	$d' = a' \cdot \frac{\partial}{\partial a}\phi_d(a, b, c) + b' \cdot \frac{\partial}{\partial b}\phi_d(a, b, c) + c' \cdot \frac{\partial}{\partial c}\phi_d(a, b, c)$
$e = \phi_e(a, b, c)$	$e' = a' \cdot \frac{\partial}{\partial a}\phi_e(a, b, c) + b' \cdot \frac{\partial}{\partial b}\phi_e(a, b, c) + c' \cdot \frac{\partial}{\partial c}\phi_e(a, b, c)$
$f = \phi_f(d, e)$	$f' = d' \cdot \frac{\partial}{\partial d}\phi_f(d, e) + e \cdot' \frac{\partial}{\partial e}\phi_d(a, b, c)$

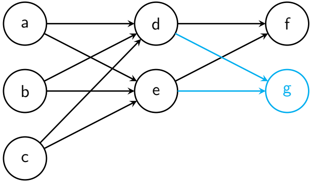
Forward mode



Let's say we want to compute $\frac{df}{da}$ with inputs i_1, i_2, i_3.
For any node x, by ϕ_x we denote the function of its inputs.
Idea: for each node x $\in$ {a,b,c,d,e,f} compute two values: x and x', where $x' = \frac{dx}{da}$.
If we want all derivatives, we need to run the algorithm for all the other input variables, so in this case 3 times.

$a = i_1$	$a' = 1$
$b = i_2$	$b' = 0$
$c = i_3$	$c' = 0$
$d = \phi_d(a, b, c)$	$d' = a' \cdot \frac{\partial}{\partial a}\phi_d(a, b, c) + b' \cdot \frac{\partial}{\partial b}\phi_d(a, b, c) + c' \cdot \frac{\partial}{\partial c}\phi_d(a, b, c)$
$e = \phi_e(a, b, c)$	$e' = a' \cdot \frac{\partial}{\partial a}\phi_e(a, b, c) + b' \cdot \frac{\partial}{\partial b}\phi_e(a, b, c) + c' \cdot \frac{\partial}{\partial c}\phi_e(a, b, c)$
$f = \phi_f(d, e)$	$f' = d' \cdot \frac{\partial}{\partial d}\phi_f(d, e) + e \cdot' \frac{\partial}{\partial e}\phi_d(a, b, c)$

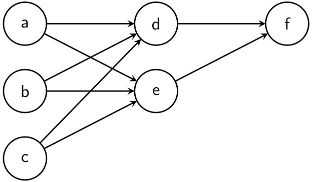
Forward mode



Let's say we want to compute $\frac{df}{da}$ with inputs i_1, i_2, i_3.
For any node x, by ϕ_x we denote the function of its inputs.
Idea: for each node x $\in$ {a,b,c,d,e,f} compute two values: x and x', where $x' = \frac{dx}{da}$.
On the other hand, if we have another output g, in one round we also compute $\frac{dg}{da}$.

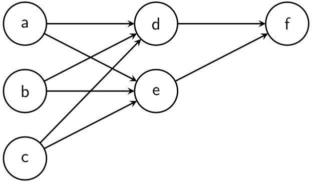
$a = i_1$	$a' = 1$
$b = i_2$	$b' = 0$
$c = i_3$	$c' = 0$
$d = \phi_d(a, b, c)$	$d' = a' \cdot \frac{\partial}{\partial a}\phi_d(a, b, c) + b' \cdot \frac{\partial}{\partial b}\phi_d(a, b, c) + c' \cdot \frac{\partial}{\partial c}\phi_d(a, b, c)$
$e = \phi_e(a, b, c)$	$e' = a' \cdot \frac{\partial}{\partial a}\phi_e(a, b, c) + b' \cdot \frac{\partial}{\partial b}\phi_e(a, b, c) + c' \cdot \frac{\partial}{\partial c}\phi_e(a, b, c)$
$f = \phi_f(d, e)$	$f' = d' \cdot \frac{\partial}{\partial d}\phi_f(d, e) + e \cdot' \frac{\partial}{\partial e}\phi_d(a, b, c)$

Reverse mode



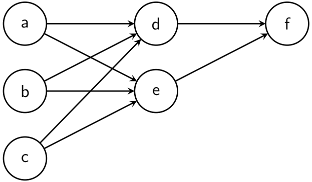
We again want to compute $\frac{df}{da}$ with inputs i_1, i_2, i_3.
For any node x, by ϕ_x we denote the function of its inputs (instead of the triangles before).

Reverse mode



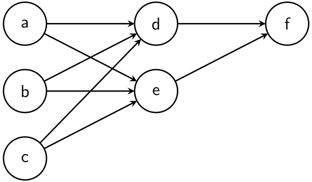
We again want to compute $\frac{df}{da}$ with inputs i_1, i_2, i_3.
For any node x, by ϕ_x we denote the function of its inputs.
Idea: for each node x $\in$ {a,b,c,d,e,f} compute two values: x and x'.

Reverse mode



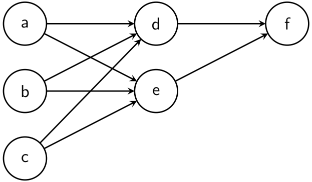
We again want to compute $\frac{df}{da}$ with inputs i_1, i_2, i_3.
For any node x, by ϕ_x we denote the function of its inputs.
Idea: for each node x $\in$ {a,b,c,d,e,f} compute two values: x and x'.
Still the value of that node

Reverse mode



We again want to compute $\frac{df}{da}$ with inputs i_1, i_2, i_3.
For any node x, by ϕ_x we denote the function of its inputs.
Idea: for each node x $\in$ {a,b,c,d,e,f} compute two values: x and x'.
This time x' is the derivative of f to that node, so $x' = \frac{df}{dx}$.

Reverse mode

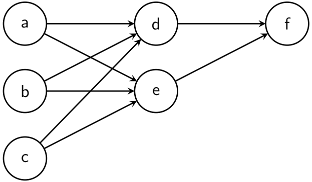


We again want to compute $\frac{df}{da}$ with inputs i_1, i_2, i_3.
For any node x, by ϕ_x we denote the function of its inputs.
Idea: for each node x $\in$ {a,b,c,d,e,f} compute two values: x and x', where $x' = \frac{df}{dx}$.
We start by initializing the starting values of the nodes without inputs (so a,b,c).

Forward pass

$a = i_1$	
$b = i_2$	
$c = i_3$	

Reverse mode

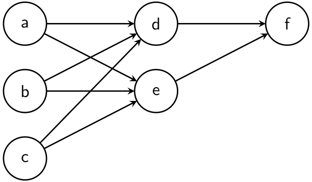


We again want to compute $\frac{df}{da}$ with inputs i_1, i_2, i_3.
For any node x, by ϕ_x we denote the function of its inputs.
Idea: for each node x $\in$ {a,b,c,d,e,f} compute two values: x and x', where $x' = \frac{df}{dx}$.
Then we can compute the value of further nodes using their functions.

Forward pass

$a = i_1$	
$b = i_2$	
$c = i_3$	
$d = \phi_d(a, b, c)$	
$e = \phi_e(a, b, c)$	
$f = \phi_f(d, e)$	

Reverse mode

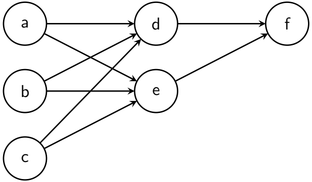


We again want to compute $\frac{df}{da}$ with inputs i_1, i_2, i_3.
For any node x, by ϕ_x we denote the function of its inputs.
Idea: for each node x $\in$ {a,b,c,d,e,f} compute two values: x and x', where $x' = \frac{df}{dx}$.
Then we start the backward pass. By definition of f', we can see it must be 1.

Backward pass

$a = i_1$	
$b = i_2$	
$c = i_3$	
$d = \phi_d(a, b, c)$	
$e = \phi_e(a, b, c)$	
$f = \phi_f(d, e)$	$f' = \frac{df}{df} = 1$

Reverse mode

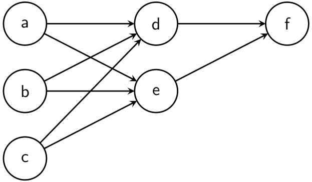


We again want to compute $\frac{df}{da}$ with inputs i_1, i_2, i_3.
For any node x, by ϕ_x we denote the function of its inputs.
Idea: for each node x $\in$ {a,b,c,d,e,f} compute two values: x and x', where $x' = \frac{df}{dx}$.
We can once again compute the derivatives using the chain rule.

Backward pass

$a = i_1$	$a' = d' \cdot \frac{\partial}{\partial a}\phi_d(d,e) + e' \cdot \frac{\partial}{\partial a}\phi_e(d,e)$
$b = i_2$	$b' = d' \cdot \frac{\partial}{\partial b}\phi_d(d,e) + e' \cdot \frac{\partial}{\partial b}\phi_e(d,e)$
$c = i_3$	$c' = d' \cdot \frac{\partial}{\partial c}\phi_d(d,e) + e' \cdot \frac{\partial}{\partial c}\phi_e(d,e)$
$d = \phi_d(a,b,c)$	$d' = f' \cdot \frac{\partial}{\partial d}\phi_f(d,e)$
$e = \phi_e(a,b,c)$	$e' = f' \cdot \frac{\partial}{\partial e}\phi_f(d,e)$
$f = \phi_f(d,e)$	$f' = \frac{df}{df} = 1$

Reverse mode

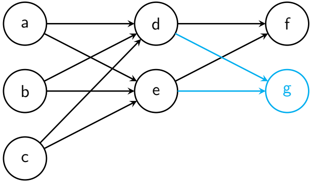


We again want to compute $\frac{df}{da}$ with inputs i_1, i_2, i_3.
For any node x, by ϕ_x we denote the function of its inputs.
Idea: for each node x $\in$ {a,b,c,d,e,f} compute two values: x and x', where $x' = \frac{df}{dx}$.
Using this algorithm, in one round we compute $\frac{df}{da}$, $\frac{df}{db}$ and $\frac{df}{da}$.

Backward pass

$a = i_1$	$a' = d' \cdot \frac{\partial}{\partial a}\phi_d(d,e) + e' \cdot \frac{\partial}{\partial a}\phi_e(d,e) = \frac{df}{da}$
$b = i_2$	$b' = d' \cdot \frac{\partial}{\partial b}\phi_d(d,e) + e' \cdot \frac{\partial}{\partial b}\phi_e(d,e) = \frac{df}{db}$
$c = i_3$	$c' = d' \cdot \frac{\partial}{\partial c}\phi_d(d,e) + e' \cdot \frac{\partial}{\partial c}\phi_e(d,e) = \frac{df}{dc}$
$d = \phi_d(a,b,c)$	$d' = f' \cdot \frac{\partial}{\partial d}\phi_f(d,e)$
$e = \phi_e(a,b,c)$	$e' = f' \cdot \frac{\partial}{\partial e}\phi_f(d,e)$
$f = \phi_f(d,e)$	$f' = \frac{df}{df} = 1$

Reverse mode



We again want to compute $\frac{df}{da}$ with inputs i_1, i_2, i_3.
For any node x, by ϕ_x we denote the function of its inputs.
Idea: for each node x ∈ {a,b,c,d,e,f} compute two values: x and x', where $x' = \frac{df}{dx}$.
On the other hand, if we have another output g, we'd need another round to also compute $\frac{dg}{da}$.

Backward pass

$a = i_1$	$a' = d' \cdot \frac{\partial}{\partial a}\phi_d(d,e) + e' \cdot \frac{\partial}{\partial a}\phi_e(d,e) = \frac{df}{da}$
$b = i_2$	$b' = d' \cdot \frac{\partial}{\partial b}\phi_d(d,e) + e' \cdot \frac{\partial}{\partial b}\phi_e(d,e) = \frac{df}{db}$
$c = i_3$	$c' = d' \cdot \frac{\partial}{\partial c}\phi_d(d,e) + e' \cdot \frac{\partial}{\partial c}\phi_e(d,e) = \frac{df}{dc}$
$d = \phi_d(a,b,c)$	$d' = f' \cdot \frac{\partial}{\partial d}\phi_f(d,e)$
$e = \phi_e(a,b,c)$	$e' = f' \cdot \frac{\partial}{\partial e}\phi_f(d,e)$
$f = \phi_f(d,e)$	$f' = \frac{df}{df} = 1$

Forward versus reverse mode: Efficiency

Let's say we are calculating the whole Jacobian of a function

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

The complexity of one round is linear in size of the computational graph $|G|$.

Forward versus reverse mode: Efficiency

Let's say we are calculating the whole Jacobian of a function

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

The complexity of one round is linear in size of the computational graph $|G|$.

1. Forward mode
   - One round per input variable.
   - Whole Jacobian is $O(n|G|)$.

Forward versus reverse mode: Efficiency

Let's say we are calculating the whole Jacobian of a function

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

The complexity of one round is linear in size of the computational graph $|G|$.

1. Forward mode
   - One round per input variable.
   - Whole Jacobian is $O(n|G|)$.
2. Reverse mode
   - One round per output variable.
   - Whole Jacobian is $O(m|G|)$.

Forward versus reverse mode: Efficiency

Let's say we are calculating the whole Jacobian of a function

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

The complexity of one round is linear in size of the computational graph $|G|$.

1. Forward mode
   - One round per input variable.
   - Whole Jacobian is $O(n|G|)$.
2. Reverse mode
   - One round per output variable.
   - Whole Jacobian is $O(m|G|)$.

Most efficient of the two is dependent on the use case. For deep learning, n can become extremely large, while $m = 1$, hence the reason why reverse mode is so widely used.

What and why

1. Program transformation: doesn't just take a mathematical function, but concrete code, and transforms it into more code that you can run.

What and why

1. Program transformation: doesn't just take a mathematical function, but concrete code, and transforms it into more code that you can run.
2. Compositional: $\overleftarrow{\mathbf{D}}(tu) = \overleftarrow{\mathbf{D}}(t)\,\overleftarrow{\mathbf{D}}(u)$.
   For example: t might be a bit of code from some external library. Using this, you only need $\overleftarrow{\mathbf{D}}(t)$ to be able to compute the derivative of your whole program.

What and why

1. Program transformation: doesn't just take a mathematical function, but concrete code, and transforms it into more code that you can run.
2. Compositional: $\overleftarrow{\mathbf{D}}(tu) = \overleftarrow{\mathbf{D}}(t)\,\overleftarrow{\mathbf{D}}(u)$.
   For example: t might be a bit of code from some external library. Using this, you only need $\overleftarrow{\mathbf{D}}(t)$ to be able to compute the derivative of your whole program.
3. Purely logical framework, allows tools from type theory, semantics etc.
   Also beneficial for soundness proof and complexity analysis

Linear negation

1. For vector space A, its dual space: linear maps $f : A \to \mathbb{R}$

Linear negation

1. For vector space A, its dual space: linear maps $f : A \to \mathbb{R}$
   We can denote this by $A^* = A \multimap \mathbb{R}$.

Linear negation

1. For vector space A, its dual space: linear maps $f : A \to \mathbb{R}$
   We can denote this by $A^* = A \multimap \mathbb{R}$.
2. Generalize to $A^{\perp d} := A \multimap \mathbb{R}^d$, the *linear negation* of A.
   We will often leave out the d.

Compositionality

Consider a simple case: the composition of two functions:
$G :=$ `let z = f x in g z` which computes $g(f(x))$.

Compositionality

Consider a simple case: the composition of two functions:
$G :=$ `let z = f x in g z` which computes $g(f(x))$.
We have the chain rule: $(g \circ f)' = (g' \circ f) \cdot g'$, which is not directly compositional.

Compositionality

Consider a simple case: the composition of two functions:
$G :=$ `let z = f x in g z` which computes $g(f(x))$.
We have the chain rule: $(g \circ f)' = (g' \circ f) \cdot g'$, which is not directly compositional.
We want some transformation $\mathbf{D}$ such that $\mathbf{D}(g \circ f) = \mathbf{D}(g) \circ \mathbf{D}(f)$ and we can retrieve f' from $\mathbf{D}(f)$.

Compositionality

Consider a simple case: the composition of two functions:
$G :=$ `let z = f x in g z` which computes $g(f(x))$.
We have the chain rule: $(g \circ f)' = (g' \circ f) \cdot g'$, which is not directly compositional.
We want some transformation $\mathbf{D}$ such that $\mathbf{D}(g \circ f) = \mathbf{D}(g) \circ \mathbf{D}(f)$ and we can retrieve f' from $\mathbf{D}(f)$.
We can use the linear negation to define such a transformation, $\mathbf{D}_{\mathrm{r}}$, where $x \in \mathbb{R}$ and $x^* \in \mathbb{R}^{\perp}$:

$$\mathbf{D}_{\mathrm{r}} f : \mathbb{R} \times \mathbb{R}^{\perp} \to \mathbb{R} \times \mathbb{R}^{\perp}$$

$$\mathbf{D}_{\mathrm{r}} f(x, x^*) := (f(x), \lambda a.x^*(f'(x) \cdot a))$$

Compositionality

Consider a simple case: the composition of two functions:
$G :=$ `let z = f x in g z` which computes $g(f(x))$.
We have the chain rule: $(g \circ f)' = (g' \circ f) \cdot g'$, which is not directly compositional.
We want some transformation $\mathbf{D}$ such that $\mathbf{D}(g \circ f) = \mathbf{D}(g) \circ \mathbf{D}(f)$ and we can retrieve f' from $\mathbf{D}(f)$.
We can use the linear negation to define such a transformation, $\mathbf{D}_{\mathrm{r}}$, where $x \in \mathbb{R}$ and $x^* \in \mathbb{R}^{\perp}$:

$$\mathbf{D}_{\mathrm{r}} f : \mathbb{R} \times \mathbb{R}^{\perp} \to \mathbb{R} \times \mathbb{R}^{\perp}$$
$$\mathbf{D}_{\mathrm{r}} f(x, x^*) := (f(x), \lambda a.x^*(f'(x) \cdot a))$$

Clearly, we can retrieve f': $(\pi_2 \mathbf{D}_{\mathrm{r}} f(x, Id))1 = f'(x)$.
But it's also compositional in the way we require.

Compositionality of $\mathbf{D}_\mathrm{r}$

$$\mathbf{D}_\mathrm{r}\, f(x, x^*) := (f(x), \lambda a.x^*(f'(x) \cdot a))$$

Expanding definition of $\mathbf{D}_\mathrm{r}\, f$

$$\mathbf{D}_\mathrm{r}\, g(\mathbf{D}_\mathrm{r}\, f(x, x^*)) = \mathbf{D}_\mathrm{r}\, g(f(x), \lambda a.x^*(f'(x) \cdot a))$$

Compositionality of $\mathbf{D}_\mathrm{r}$

$$\mathbf{D}_\mathrm{r}\, f(x, x^*) := (f(x), \lambda a.x^*(f'(x) \cdot a))$$

Expanding definition of $\mathbf{D}_\mathrm{r}\, g$

$$\begin{aligned} \mathbf{D}_\mathrm{r}\, g(\mathbf{D}_\mathrm{r}\, f(x, x^*)) &= \mathbf{D}_\mathrm{r}\, g(f(x), \lambda a.x^*(f'(x) \cdot a)) \\ &= (g(f(x)), \lambda b.(\lambda a.x^*(f'(x) \cdot a))(g'(f(x)) \cdot b)) \end{aligned}$$

Compositionality of $\mathbf{D}_{\mathrm{r}}$

$$\mathbf{D}_{\mathrm{r}}\, f(x, x^*) := (f(x), \lambda a.x^*(f'(x) \cdot a))$$

β-reduction

$$\begin{aligned}
\mathbf{D}_{\mathrm{r}}\, g(\mathbf{D}_{\mathrm{r}}\, f(x, x^*)) &= \mathbf{D}_{\mathrm{r}}\, g(f(x), \lambda a.x^*(f'(x) \cdot a)) \\
&= (g(f(x)), \lambda b.(\lambda a.x^*(f'(x) \cdot a))(g'(f(x)) \cdot b)) \\
&= (g(f(x)), \lambda b.x^*(f'(x) \cdot (g'(f(x)) \cdot b))
\end{aligned}$$

Compositionality of $\mathbf{D}_\mathrm{r}$

$$\mathbf{D}_\mathrm{r}\, f(x, x^*) := (f(x), \lambda a.x^*(f'(x)\cdot a))$$

Reordering terms

$$\begin{aligned}\mathbf{D}_\mathrm{r}\, g(\mathbf{D}_\mathrm{r}\, f(x,x^*)) &= \mathbf{D}_\mathrm{r}\, g(f(x), \lambda a.x^*(f'(x)\cdot a)) \\ &= (g(f(x)), \lambda b.(\lambda a.x^*(f'(x)\cdot a))(g'(f(x))\cdot b)) \\ &= (g(f(x)), \lambda b.x^*(f'(x)\cdot (g'(f(x))\cdot b)) \\ &= (g(f(x)), \lambda b.x^*((g'(f(x))\cdot f'(x))\cdot b))\end{aligned}$$

Compositionality of $\mathbf{D}_\mathrm{r}$

$$\mathbf{D}_\mathrm{r}\, f(x, x^*) := (f(x), \lambda a. x^*(f'(x) \cdot a))$$

Chain rule and contracting the composition of g and f.

$$\begin{aligned}
\mathbf{D}_\mathrm{r}\, g(\mathbf{D}_\mathrm{r}\, f(x, x^*)) &= \mathbf{D}_\mathrm{r}\, g(f(x), \lambda a. x^*(f'(x) \cdot a)) \\
&= (g(f(x)), \lambda b.(\lambda a. x^*(f'(x) \cdot a))(g'(f(x)) \cdot b)) \\
&= (g(f(x)), \lambda b. x^*(f'(x) \cdot (g'(f(x)) \cdot b)) \\
&= (g(f(x)), \lambda b. x^*((g'(f(x)) \cdot f'(x)) \cdot b)) \\
&= ((g \circ f)(x), \lambda b. x^*((g \circ f)'(x) \cdot b))
\end{aligned}$$

Compositionality of $\mathbf{D}_\mathrm{r}$

$$\mathbf{D}_\mathrm{r}\, f(x, x^*) := (f(x), \lambda a.x^*(f'(x) \cdot a))$$

Definition of $\mathbf{D}_\mathrm{r}$

$$\begin{aligned}
\mathbf{D}_\mathrm{r}\, g(\mathbf{D}_\mathrm{r}\, f(x, x^*)) &= \mathbf{D}_\mathrm{r}\, g(f(x), \lambda a.x^*(f'(x) \cdot a)) \\
&= (g(f(x)), \lambda b.(\lambda a.x^*(f'(x) \cdot a))(g'(f(x)) \cdot b)) \\
&= (g(f(x)), \lambda b.x^*(f'(x) \cdot (g'(f(x)) \cdot b)) \\
&= (g(f(x)), \lambda b.x^*((g'(f(x)) \cdot f'(x)) \cdot b)) \\
&= ((g \circ f)(x), \lambda b.x^*((g \circ f)'(x) \cdot b)) \\
&= \mathbf{D}_\mathrm{r}(g \circ f)(x, x^*).
\end{aligned}$$

Generalizing $\mathbf{D}_\mathrm{r}$

$$\mathbf{D}_\mathrm{r}\, f(x, x^*) := (f(x), \lambda a.x^*(f'(x) \cdot a))$$

We can generalize this one-dimensional transformation to maps $f : \mathbb{R}^n \to \mathbb{R}$, where $\mathbf{x} \in \mathbb{R}^n$ and $\mathbf{x}^* = (x_1^* \ldots x_n^*) \in (\mathbb{R}^\perp)^n$:

Generalizing $\mathbf{D}_{\mathrm{r}}$

$$\mathbf{D}_{\mathrm{r}}\, f(x, x^*) := (f(x), \lambda a. x^*(f'(x) \cdot a))$$

We can generalize this one-dimensional transformation to maps $f : \mathbb{R}^n \to \mathbb{R}$, where $\mathbf{x} \in \mathbb{R}^n$ and $\mathbf{x}^* = (x_1^* \ldots x_n^*) \in (\mathbb{R}^{\perp})^n$:

$$\overleftarrow{\mathbf{D}}(f)(\mathbf{x}, \mathbf{x}^*) = \left(f(\mathbf{x}), \lambda a. \sum_{i=1}^{n} x_i^*(\partial_i f(\mathbf{x}) \cdot a) \right)$$

Generalizing $\mathbf{D}_r$

$$\mathbf{D}_r\, f(x, x^*) := (f(x), \lambda a. x^*(f'(x) \cdot a))$$

We can generalize this one-dimensional transformation to maps $f : \mathbb{R}^n \to \mathbb{R}$, where $\mathbf{x} \in \mathbb{R}^n$ and $\mathbf{x}^* = (x_1^* \ldots x_n^*) \in (\mathbb{R}^\perp)^n$:

$$\overleftarrow{\mathbf{D}}(f)(\mathbf{x}, \mathbf{x}^*) = \left(f(\mathbf{x}), \lambda a. \sum_{i=1}^{n} x_i^*(\partial_i f(\mathbf{x}) \cdot a) \right)$$

Now we want to go one step further, and define a compositional program transformation that does the same.

Linear substitution algebra

Based on simply typed λ-calculus, but with the addition of linear negation.

Linear substitution algebra: types and grammar

$$
\begin{aligned}
A, B, C &::= R \mid A \times B \mid A \to B \mid R^{\perp d} && \text{(types)} \\
v &::= x^{(!)A} \mid \underline{r} \mid \lambda x^{(!)A}.t \mid \langle v_1, v_2 \rangle && \text{(values)} \\
t, u &::= v \mid tu \mid \langle t, u \rangle \mid t[\langle x^{!A}, y^{!B} \rangle := u] \\
&\quad \mid t[x^{(!)A} := u] \mid t + u \mid f(t_1, \ldots, t_k) && \text{(terms)}
\end{aligned}
$$

$R^{\perp d}$ is the type representing the linear negation of R for some $d \in \mathbb{N}$.

Linear substitution algebra: types and grammar

$$A, B, C ::= R \mid A \times B \mid A \to B \mid R^{\perp d} \qquad \text{(types)}$$

$$v ::= x^{(!)A} \mid \underline{r} \mid \lambda x^{(!)A}.t \mid \langle v_1, v_2 \rangle \qquad \text{(values)}$$

$$t, u ::= v \mid tu \mid \langle t, u \rangle \mid t[\langle x^{!A}, y^{!B} \rangle := u]$$

$$\mid t[x^{(!)A} := u] \mid t + u \mid f(t_1, \ldots, t_k) \qquad \text{(terms)}$$

$x^{(!)A}$ ranges over annotated variables, either *exponential variables* of any type A: $x^{!A}$, or *linear variables* specifically of type R: x^R.

Linear substitution algebra: types and grammar

$$A, B, C ::= R \mid A \times B \mid A \to B \mid R^{\perp d} \qquad \text{(types)}$$

$$v ::= x^{(!)A} \mid \underline{r} \mid \lambda x^{(!)A}.t \mid \langle v_1, v_2 \rangle \qquad \text{(values)}$$

$$t, u ::= v \mid tu \mid \langle t, u \rangle \mid t[\langle x^{!A}, y^{!B} \rangle := u]$$
$$\mid t[x^{(!)A} := u] \mid t + u \mid f(t_1, \ldots, t_k) \qquad \text{(terms)}$$

These denote substitution, more familiar in the form `let x = u in t`, and its binary variant.

Linear substitution algebra: types and grammar

$$
\begin{aligned}
A, B, C &::= R \mid A \times B \mid A \to B \mid R^{\perp d} && \text{(types)} \\
v &::= x^{(!)A} \mid \underline{r} \mid \lambda x^{(!)A}.t \mid \langle v_1, v_2 \rangle && \text{(values)} \\
t, u &::= v \mid tu \mid \langle t, u \rangle \mid t[\langle x^{!A}, y^{!B} \rangle := u] \\
&\quad \mid t[x^{(!)A} := u] \mid t + u \mid f(t_1, \ldots, t_k) && \text{(terms)}
\end{aligned}
$$

f ranges over the function symbols $\mathcal{F}$, including at least multiplication $t_1 \cdot t_2$

Linear substitution algebra: Typing rules

$$\frac{}{\Gamma \vdash_z z : \mathrm{R}} \qquad \frac{}{\Gamma, x^{!A} \vdash x : A} \qquad \frac{\Gamma \vdash_{(z)} t : A \quad \Gamma \vdash_{(z)} u : B}{\Gamma \vdash_{(z)} \langle t, u \rangle : A \times B} \qquad \frac{\Gamma \vdash u : A \times B \quad \Gamma, x^{!A}, y^{!B} \vdash_{(z)} t : C}{\Gamma \vdash_{(z)} t[\langle x^{!A}, y^{!B} \rangle := u] : C}$$

$$\frac{\Gamma, x^{!A} \vdash t : B}{\Gamma \vdash \lambda x^{!A}.t : A \to B} \qquad \frac{\Gamma \vdash t : A \to B \quad \Gamma \vdash u : A}{\Gamma \vdash tu : B} \qquad \frac{\Gamma \vdash_z t : \mathrm{R}^d}{\Gamma \vdash \lambda z^{\mathrm{R}}.t : \mathrm{R}^{\perp d}} \qquad \frac{\Gamma \vdash t : \mathrm{R}^{\perp d} \quad \Gamma \vdash_{(z)} u : \mathrm{R}}{\Gamma \vdash_{(z)} tu : \mathrm{R}^d}$$

$$\frac{\Gamma \vdash u : A \quad \Gamma, x^{!A} \vdash_{(z)} t : C}{\Gamma \vdash_{(z)} t[x^{!A} := u] : C} \qquad \frac{\Gamma \vdash_{(z')} u : \mathrm{R} \quad \Gamma \vdash_z t : \mathrm{R}^d}{\Gamma \vdash_{(z')} t[z^{\mathrm{R}} := u] : \mathrm{R}^d} \qquad \frac{\Gamma \vdash t_1 : \mathrm{R} \quad \dots \quad \Gamma \vdash t_k : \mathrm{R}}{\Gamma \vdash f(t_1, \dots, t_k) : \mathrm{R}} \qquad \frac{r \in \mathbb{R}}{\Gamma \vdash \underline{r} : \mathrm{R}}$$

$$\frac{\Gamma \vdash_{(z)} t : \mathrm{R} \quad \Gamma \vdash u : \mathrm{R}}{\Gamma \vdash_{(z)} t \cdot u : \mathrm{R}} \qquad \frac{\Gamma \vdash t : \mathrm{R} \quad \Gamma \vdash_{(z)} u : \mathrm{R}}{\Gamma \vdash_{(z)} t \cdot u : \mathrm{R}} \qquad \frac{}{\Gamma \vdash_z \underline{0} : \mathrm{R}^d} \qquad \frac{\Gamma \vdash_{(z)} t : \mathrm{R}^d \quad \Gamma \vdash_{(z)} u : \mathrm{R}^d}{\Gamma \vdash_{(z)} t + u : \mathrm{R}^d}$$

Fig. 3. The typing rules. In the pairing and sum rules, either all three sequents have z, or none does.

Linear substitution algebra: Typing rules

$$\frac{}{\Gamma \vdash_z z : \mathrm{R}} \qquad \frac{}{\Gamma, x^{!A} \vdash x : A} \qquad \frac{\Gamma \vdash_{(z)} t : A \quad \Gamma \vdash_{(z)} u : B}{\Gamma \vdash_{(z)} \langle t, u \rangle : A \times B} \qquad \frac{\Gamma \vdash u : A \times B \quad \Gamma, x^{!A}, y^{!B} \vdash_{(z)} t : C}{\Gamma \vdash_{(z)} t[\langle x^{!A}, y^{!B} \rangle := u] : C}$$

$$\frac{\Gamma, x^{!A} \vdash t : B}{\Gamma \vdash \lambda x^{!A}.t : A \to B} \qquad \frac{\Gamma \vdash t : A \to B \quad \Gamma \vdash u : A}{\Gamma \vdash tu : B} \qquad \frac{\Gamma \vdash_z t : \mathrm{R}^d}{\Gamma \vdash \lambda z^{\mathrm{R}}.t : \mathrm{R}^{\perp d}} \qquad \frac{\Gamma \vdash t : \mathrm{R}^{\perp d} \quad \Gamma \vdash_{(z)} u : \mathrm{R}}{\Gamma \vdash_{(z)} tu : \mathrm{R}^d}$$

$$\frac{\Gamma \vdash u : A \quad \Gamma, x^{!A} \vdash_{(z)} t : C}{\Gamma \vdash_{(z)} t[x^{!A} := u] : C} \qquad \frac{\Gamma \vdash_{(z')} u : \mathrm{R} \quad \Gamma \vdash_z t : \mathrm{R}^d}{\Gamma \vdash_{(z')} t[z^{\mathrm{R}} := u] : \mathrm{R}^d} \qquad \frac{\Gamma \vdash t_1 : \mathrm{R} \quad \ldots \quad \Gamma \vdash t_k : \mathrm{R}}{\Gamma \vdash f(t_1, \ldots, t_k) : \mathrm{R}} \qquad \frac{r \in \mathbb{R}}{\Gamma \vdash \underline{r} : \mathrm{R}}$$

$$\frac{\Gamma \vdash_{(z)} t : \mathrm{R} \quad \Gamma \vdash u : \mathrm{R}}{\Gamma \vdash_{(z)} t \cdot u : \mathrm{R}} \qquad \frac{\Gamma \vdash t : \mathrm{R} \quad \Gamma \vdash_{(z)} u : \mathrm{R}}{\Gamma \vdash_{(z)} t \cdot u : \mathrm{R}} \qquad \frac{}{\Gamma \vdash_z \underline{0} : \mathrm{R}^d} \qquad \frac{\Gamma \vdash_{(z)} t : \mathrm{R}^d \quad \Gamma \vdash_{(z)} u : \mathrm{R}^d}{\Gamma \vdash_{(z)} t + u : \mathrm{R}^d}$$

Fig. 3. The typing rules. In the pairing and sum rules, either all three sequents have z, or none does.

Two types of sequents: $\Gamma \vdash t : A$ and $\Gamma \vdash_z t : R_d$

Linear substitution algebra: Typing rules

Two types of sequents: $\Gamma \vdash t : A$ and $\Gamma \vdash_z t : R_d$

$$\frac{\Gamma \ \vdash_z t : R^d}{\Gamma \vdash \lambda z^R.t : R^{\perp d}} \qquad \frac{}{\Gamma \ \vdash_z z : R} \qquad \frac{\Gamma \ \vdash_{(z)} t : R \qquad \Gamma \vdash u : R}{\Gamma \ \vdash_{(z)} t \cdot u : R}$$

z in $\Gamma \vdash_z t : R_d$ is a linear type annotated variable which occurs free *linearly* in t.
Some rules exist for both sequents.

Linear substitution algebra: Typing rules

Two types of sequents: $\Gamma \vdash t : A$ and $\Gamma \vdash_z t : R_d$

$$\frac{\Gamma \vdash_z t : R^d}{\Gamma \vdash \lambda z^R . t : R^{\perp d}} \qquad \frac{}{\Gamma \vdash_z z : R} \qquad \frac{\Gamma \vdash_{(z)} t : R \qquad \Gamma \vdash u : R}{\Gamma \vdash_{(z)} t \cdot u : R}$$

This is the typing rule for the linear negation. Linear variable z must occur linearly in t, then the lambda that binds z is a linear map.

Linear substitution algebra: Typing rules

Two types of sequents: $\Gamma \vdash t : A$ and $\Gamma \vdash_z t : R_d$

$$\frac{\Gamma \vdash_z t : R^d}{\Gamma \vdash \lambda z^R . t : R^{\perp d}} \qquad \frac{}{\Gamma \vdash_z z : R} \qquad \frac{\Gamma \vdash_{(z)} t : R \qquad \Gamma \vdash u : R}{\Gamma \vdash_{(z)} t \cdot u : R}$$

These are some of the rules that showcase what it means for z to occur linearly.

The program transformation $\overleftarrow{\mathbf{D}}_d$

$$\overleftarrow{\mathbf{D}}_d(x^{!A}) := x^{!\overleftarrow{\mathbf{D}}_d(A)}$$

$$\overleftarrow{\mathbf{D}}_d(\lambda x^{!A}.t) := \lambda x^{!\overleftarrow{\mathbf{D}}_d(A)}.\overleftarrow{\mathbf{D}}_d(t)$$

$$\overleftarrow{\mathbf{D}}_d(tu) := \overleftarrow{\mathbf{D}}_d(t)\overleftarrow{\mathbf{D}}_d(u)$$

$$\overleftarrow{\mathbf{D}}_d(\langle t, u\rangle) := \left\langle \overleftarrow{\mathbf{D}}_d(t), \overleftarrow{\mathbf{D}}_d(u) \right\rangle$$

$$\overleftarrow{\mathbf{D}}_d(t[\langle x^{!A}, y^{!B}\rangle := u]) := \overleftarrow{\mathbf{D}}_d(t)[\left\langle x^{!\overleftarrow{\mathbf{D}}_d(A)}, y^{!\overleftarrow{\mathbf{D}}_d(B)} \right\rangle := \overleftarrow{\mathbf{D}}_d(u)]$$

$$\overleftarrow{\mathbf{D}}_d(t[x^{!A} := u]) := \overleftarrow{\mathbf{D}}_d(t)[x^{!\overleftarrow{\mathbf{D}}_d(A)} := \overleftarrow{\mathbf{D}}_d(u)]$$

$$\overleftarrow{\mathbf{D}}_d(\underline{r}) := \langle \underline{r}, \lambda a^{\mathrm{R}}.\underline{\mathbf{0}}\rangle$$

$$\overleftarrow{\mathbf{D}}_d(t + u) := \langle x + y, \lambda a^{\mathrm{R}}.(x^* a + y^* a)\rangle \left[\left\langle x^{!\mathrm{R}}, x^{*!\mathrm{R}^{\perp d}} \right\rangle := \overleftarrow{\mathbf{D}}_d(t)\right]\left[\left\langle y^{!\mathrm{R}}, y^{*!\mathrm{R}^{\perp d}} \right\rangle := \overleftarrow{\mathbf{D}}_d(u)\right]$$

$$\overleftarrow{\mathbf{D}}_d(f(\mathbf{t})) := \left\langle f(\mathbf{x})\,,\ \lambda a^{\mathrm{R}}.\sum_{i=1}^{k} x_i^* \left(\partial_i f(\mathbf{x}) \cdot a\right) \right\rangle \left[\left\langle \mathbf{x}^{!\mathrm{R}}, \mathbf{x}^{*!\mathrm{R}^{\perp d}} \right\rangle := \overleftarrow{\mathbf{D}}_d(\mathbf{t})\right]$$

$$\overleftarrow{\mathbf{D}}_d(\mathrm{R}) := \mathrm{R} \times \mathrm{R}^{\perp d}$$

$$\overleftarrow{\mathbf{D}}_d(A \to B) := \overleftarrow{\mathbf{D}}_d(A) \to \overleftarrow{\mathbf{D}}_d(B)$$

$$\overleftarrow{\mathbf{D}}_d(A \times B) := \overleftarrow{\mathbf{D}}_d(A) \times \overleftarrow{\mathbf{D}}_d(B)$$

The program transformation $\overleftarrow{\mathbf{D}}_d$

$$
\begin{aligned}
\overleftarrow{\mathbf{D}}_d(x^{!A}) &:= x^{!\overleftarrow{\mathbf{D}}_d(A)} \\
\overleftarrow{\mathbf{D}}_d(\lambda x^{!A}.t) &:= \lambda x^{!\overleftarrow{\mathbf{D}}_d(A)}.\overleftarrow{\mathbf{D}}_d(t) \\
\overleftarrow{\mathbf{D}}_d(tu) &:= \overleftarrow{\mathbf{D}}_d(t)\overleftarrow{\mathbf{D}}_d(u) \\
\overleftarrow{\mathbf{D}}_d(\langle t, u\rangle) &:= \left\langle \overleftarrow{\mathbf{D}}_d(t), \overleftarrow{\mathbf{D}}_d(u) \right\rangle \\
\overleftarrow{\mathbf{D}}_d(t[\langle x^{!A}, y^{!B}\rangle := u]) &:= \overleftarrow{\mathbf{D}}_d(t)[\left\langle x^{!\overleftarrow{\mathbf{D}}_d(A)}, y^{!\overleftarrow{\mathbf{D}}_d(B)} \right\rangle := \overleftarrow{\mathbf{D}}_d(u)] \\
\overleftarrow{\mathbf{D}}_d(t[x^{!A} := u]) &:= \overleftarrow{\mathbf{D}}_d(t)[x^{!\overleftarrow{\mathbf{D}}_d(A)} := \overleftarrow{\mathbf{D}}_d(u)] \\
\overleftarrow{\mathbf{D}}_d(\underline{r}) &:= \langle \underline{r}, \lambda a^{\mathrm{R}}.\underline{\mathbf{0}} \rangle \\
\overleftarrow{\mathbf{D}}_d(t+u) &:= \langle x+y, \lambda a^{\mathrm{R}}.(x^* a + y^* a)\rangle\, [\left\langle x^{!\mathrm{R}}, x^{*!\mathrm{R}^{\perp d}} \right\rangle := \overleftarrow{\mathbf{D}}_d(t)][\left\langle y^{!\mathrm{R}}, y^{*!\mathrm{R}^{\perp d}} \right\rangle := \overleftarrow{\mathbf{D}}_d(u)] \\
\overleftarrow{\mathbf{D}}_d(f(\mathbf{t})) &:= \left\langle f(\mathbf{x})\,,\ \lambda a^{\mathrm{R}}.\sum_{i=1}^{k} x_i^* \left(\partial_i f(\mathbf{x}) \cdot a\right) \right\rangle [\left\langle \mathbf{x}^{!\mathrm{R}}, \mathbf{x}^{*!\mathrm{R}^{\perp d}} \right\rangle := \overleftarrow{\mathbf{D}}_d(\mathbf{t})]
\end{aligned}
$$

$$
\begin{aligned}
\overleftarrow{\mathbf{D}}_d(\mathrm{R}) &:= \mathrm{R} \times \mathrm{R}^{\perp d} \\
\overleftarrow{\mathbf{D}}_d(A \to B) &:= \overleftarrow{\mathbf{D}}_d(A) \to \overleftarrow{\mathbf{D}}_d(B) \\
\overleftarrow{\mathbf{D}}_d(A \times B) &:= \overleftarrow{\mathbf{D}}_d(A) \times \overleftarrow{\mathbf{D}}_d(B)
\end{aligned}
$$

The program transformation $\overleftarrow{\mathbf{D}}_d$

$$\overleftarrow{\mathbf{D}}_d(x^{!A}) := x^{!\overleftarrow{\mathbf{D}}_d(A)}$$
$$\overleftarrow{\mathbf{D}}_d(\lambda x^{!A}.t) := \lambda x^{!\overleftarrow{\mathbf{D}}_d(A)}.\overleftarrow{\mathbf{D}}_d(t)$$
$$\overleftarrow{\mathbf{D}}_d(tu) := \overleftarrow{\mathbf{D}}_d(t)\overleftarrow{\mathbf{D}}_d(u)$$
$$\overleftarrow{\mathbf{D}}_d(\langle t,u\rangle) := \left\langle \overleftarrow{\mathbf{D}}_d(t), \overleftarrow{\mathbf{D}}_d(u)\right\rangle$$
$$\overleftarrow{\mathbf{D}}_d(t[\left\langle x^{!A}, y^{!B}\right\rangle := u]) := \overleftarrow{\mathbf{D}}_d(t)[\left\langle x^{!\overleftarrow{\mathbf{D}}_d(A)}, y^{!\overleftarrow{\mathbf{D}}_d(B)}\right\rangle := \overleftarrow{\mathbf{D}}_d(u)]$$
$$\overleftarrow{\mathbf{D}}_d(t[x^{!A} := u]) := \overleftarrow{\mathbf{D}}_d(t)[x^{!\overleftarrow{\mathbf{D}}_d(A)} := \overleftarrow{\mathbf{D}}_d(u)]$$
$$\overleftarrow{\mathbf{D}}_d(\underline{r}) := \langle \underline{r}, \lambda a^{\mathrm{R}}.\underline{\mathbf{0}}\rangle$$

$$\overleftarrow{\mathbf{D}}_d(\mathrm{R}) := \mathrm{R} \times \mathrm{R}^{\perp d}$$
$$\overleftarrow{\mathbf{D}}_d(A \to B) := \overleftarrow{\mathbf{D}}_d(A) \to \overleftarrow{\mathbf{D}}_d(B)$$
$$\overleftarrow{\mathbf{D}}_d(A \times B) := \overleftarrow{\mathbf{D}}_d(A) \times \overleftarrow{\mathbf{D}}_d(B)$$

$$\overleftarrow{\mathbf{D}}_d(t+u) := \langle x+y, \lambda a^{\mathrm{R}}.(x^* a + y^* a)\rangle\,[\left\langle x^{!\mathrm{R}}, x^{*!\mathrm{R}^{\perp d}}\right\rangle := \overleftarrow{\mathbf{D}}_d(t)][\left\langle y^{!\mathrm{R}}, y^{*!\mathrm{R}^{\perp d}}\right\rangle := \overleftarrow{\mathbf{D}}_d(u)]$$

$$\overleftarrow{\mathbf{D}}_d(f(\mathbf{t})) := \left\langle f(\mathbf{x})\,,\ \lambda a^{\mathrm{R}}.\sum_{i=1}^{k} x_i^*\,(\partial_i f(\mathbf{x}) \cdot a)\right\rangle [\left\langle \mathbf{x}^{!\mathrm{R}}, \mathbf{x}^{*!\mathrm{R}^{\perp d}}\right\rangle := \overleftarrow{\mathbf{D}}_d(\mathbf{t})]$$

The program transformation $\overleftarrow{\mathrm{D}}_d$

$$\overleftarrow{\mathrm{D}}_d(x^{!A}) := x^{!\overleftarrow{\mathrm{D}}_d(A)}$$
$$\overleftarrow{\mathrm{D}}_d(\lambda x^{!A}.t) := \lambda x^{!\overleftarrow{\mathrm{D}}_d(A)}.\overleftarrow{\mathrm{D}}_d(t)$$
$$\overleftarrow{\mathrm{D}}_d(tu) := \overleftarrow{\mathrm{D}}_d(t)\overleftarrow{\mathrm{D}}_d(u)$$
$$\overleftarrow{\mathrm{D}}_d(\langle t, u\rangle) := \left\langle \overleftarrow{\mathrm{D}}_d(t), \overleftarrow{\mathrm{D}}_d(u)\right\rangle$$
$$\overleftarrow{\mathrm{D}}_d(t[\langle x^{!A}, y^{!B}\rangle := u]) := \overleftarrow{\mathrm{D}}_d(t)[\left\langle x^{!\overleftarrow{\mathrm{D}}_d(A)}, y^{!\overleftarrow{\mathrm{D}}_d(B)}\right\rangle := \overleftarrow{\mathrm{D}}_d(u)]$$
$$\overleftarrow{\mathrm{D}}_d(t[x^{!A} := u]) := \overleftarrow{\mathrm{D}}_d(t)[x^{!\overleftarrow{\mathrm{D}}_d(A)} := \overleftarrow{\mathrm{D}}_d(u)]$$
$$\overleftarrow{\mathrm{D}}_d(\underline{r}) := \langle \underline{r}, \lambda a^{\mathrm{R}}.\underline{\mathbf{0}}\rangle$$
$$\overleftarrow{\mathrm{D}}_d(t+u) := \langle x+y, \lambda a^{\mathrm{R}}.(x^* a + y^* a)\rangle \left[\left\langle x^{!\mathrm{R}}, x^{*!\mathrm{R}^{\perp d}}\right\rangle := \overleftarrow{\mathrm{D}}_d(t)\right]\left[\left\langle y^{!\mathrm{R}}, y^{*!\mathrm{R}^{\perp d}}\right\rangle := \overleftarrow{\mathrm{D}}_d(u)\right]$$
$$\overleftarrow{\mathrm{D}}_d(f(\mathbf{t})) := \left\langle f(\mathbf{x})\,,\ \lambda a^{\mathrm{R}}.\sum_{i=1}^{k} x_i^*\,(\partial_i f(\mathbf{x})\cdot a)\right\rangle \left[\left\langle \mathbf{x}^{!\mathrm{R}}, \mathbf{x}^{*!\mathrm{R}^{\perp d}}\right\rangle := \overleftarrow{\mathrm{D}}_d(\mathbf{t})\right]$$

$$\overleftarrow{\mathrm{D}}_d(\mathrm{R}) := \mathrm{R}\times\mathrm{R}^{\perp d}$$
$$\overleftarrow{\mathrm{D}}_d(A\to B) := \overleftarrow{\mathrm{D}}_d(A)\to\overleftarrow{\mathrm{D}}_d(B)$$
$$\overleftarrow{\mathrm{D}}_d(A\times B) := \overleftarrow{\mathrm{D}}_d(A)\times\overleftarrow{\mathrm{D}}_d(B)$$

Example

$\sin(x \cdot x) \longrightarrow$ `let z = x*x in sin z`

$$\overleftarrow{\mathbf{D}}(\sin(z^{!R})[z^{!R} := x \cdot x])$$

Example

$\sin(x \cdot x) \longrightarrow$ `let z = x*x in sin z`

$$\overleftarrow{\mathbf{D}}(\sin(z^{!R})[z^{!R} := x \cdot x])$$
$$= \overleftarrow{\mathbf{D}}(\sin(z^{!R}))[z^{!R \times R^{\perp}} := \overleftarrow{\mathbf{D}}(x \cdot x)]$$

Example

$\sin(x \cdot x) \longrightarrow$ `let z = x*x in sin z`

$$\overleftarrow{\mathbf{D}}(\sin(z^{!R})[z^{!R} := x \cdot x])$$

$$= \overleftarrow{\mathbf{D}}(\sin(z^{!R}))[z^{!R \times R^{\perp}} := \overleftarrow{\mathbf{D}}(x \cdot x)]$$

$$= \langle \sin(t), \lambda a^{R}.t^{*}(\cos(t) \cdot a)\rangle[\langle t^{!R}, t^{*!R^{\perp}}\rangle := z^{!R \times R^{\perp}}][z^{!R \times R^{\perp}} := \overleftarrow{\mathbf{D}}(x \cdot x)]$$

Example

$\sin(x \cdot x) \longrightarrow$ `let z = x*x in sin z`

$$
\begin{aligned}
\overleftarrow{\mathbf{D}}(\sin(z^{!R})[z^{!R} := x \cdot x]) \\
&= \overleftarrow{\mathbf{D}}(\sin(z^{!R}))[z^{!R \times R^{\perp}} := \overleftarrow{\mathbf{D}}(x \cdot x)] \\
&= \langle \sin(t), \lambda a^{R}.t^{*}(\cos(t) \cdot a)\rangle[\langle t^{!R}, t^{*!R^{\perp}}\rangle := z^{!R \times R^{\perp}}][z^{!R \times R^{\perp}} := \overleftarrow{\mathbf{D}}(x \cdot x)] \\
&= \langle \sin(t), \lambda a^{R}.t^{*}(\cos(t) \cdot a)\rangle[\langle t^{!R}, t^{*!R^{\perp}}\rangle := z^{!R \times R^{\perp}}] \\
&\quad [z^{!R \times R^{\perp}} := \langle s \cdot s, \lambda b^{R}.s^{*}((s+s) \cdot b)\rangle[\langle s^{!R}, s^{*!R^{\perp}}\rangle := x^{!R \times R^{\perp}}]]
\end{aligned}
$$

Example

$\sin(x \cdot x) \longrightarrow$ `let z = x*x in sin z`

$$
\begin{aligned}
\overleftarrow{\mathbf{D}}(\sin(z^{!R})[z^{!R} := x \cdot x]) \\
&= \overleftarrow{\mathbf{D}}(\sin(z^{!R}))[z^{!R \times R^{\perp}} := \overleftarrow{\mathbf{D}}(x \cdot x)] \\
&= \langle \sin(t), \lambda a^{R}.t^{*}(\cos(t) \cdot a)\rangle[\langle t^{!R}, t^{*!R^{\perp}}\rangle := z^{!R \times R^{\perp}}][z^{!R \times R^{\perp}} := \overleftarrow{\mathbf{D}}(x \cdot x)] \\
&= \langle \sin(t), \lambda a^{R}.t^{*}(\cos(t) \cdot a)\rangle[\langle t^{!R}, t^{*!R^{\perp}}\rangle := z^{!R \times R^{\perp}}] \\
&\quad [z^{!R \times R^{\perp}} := \langle s \cdot s, \lambda b^{R}.s^{*}((s+s) \cdot b)\rangle[\langle s^{!R}, s^{*!R^{\perp}}\rangle := x^{!R \times R^{\perp}}]]
\end{aligned}
$$

Example

$\sin(x \cdot x) \longrightarrow$ `let z = x*x in sin z`

$$\overleftarrow{\mathbf{D}}(\sin(z^{!R})[z^{!R} := x \cdot x])$$

$$= \overleftarrow{\mathbf{D}}(\sin(z^{!R}))[z^{!R \times R^\perp} := \overleftarrow{\mathbf{D}}(x \cdot x)]$$

$$= \langle \sin(t), \lambda a^R . t^*(\cos(t) \cdot a)\rangle[\langle t^{!R}, t^{*!R^\perp}\rangle := z^{!R \times R^\perp}][z^{!R \times R^\perp} := \overleftarrow{\mathbf{D}}(x \cdot x)]$$

$$= \langle \sin(t), \lambda a^R . t^*(\cos(t) \cdot a)\rangle[\langle t^{!R}, t^{*!R^\perp}\rangle := z^{!R \times R^\perp}]$$

$$[z^{!R \times R^\perp} := \langle s \cdot s, \lambda b^R . s^*((s + s) \cdot b)\rangle[\langle s^{!R}, s^{*!R^\perp}\rangle := x^{!R \times R^\perp}]]$$

We can extract the derivative as before, yielding:

$$(\pi_2 \overleftarrow{\mathbf{D}}(\sin(z^{!R})[z^{!R} := x \cdot x])(i, Id))1 = (i + i) \cdot \cos(i \cdot i) = \left(\frac{d}{dx} \sin(x \cdot x)\right)(i)$$

The End