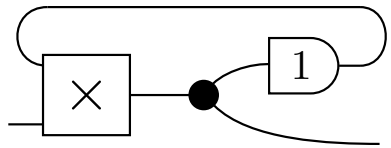
Differentiation of causal functions

Shin-ya Katsumata and David Sprunger*
National Institute of Informatics
ERATO MMSD

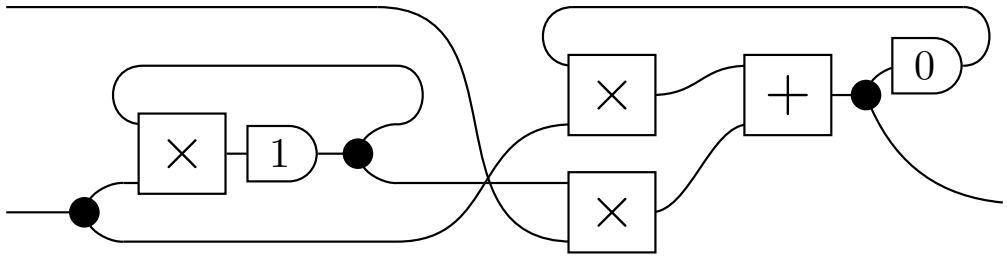
IU Logic Colloquium
March 21, 2019

Where we're heading

I would like to convince you the derivative of this function:



Is this:



How we get there

1. Causal functions
2. Computation sequences
   - Main ideas
   - Sanity check
3. Delayed trace
4. Neural networks
5. Cartesian differential categories
6. Recap and future directions

Causal functions on sequences

A^ω is the set of *A-valued infinite sequences*. The entries of $\sigma \in A^\omega$ are $[\sigma]_k = \sigma_k \in A$ for $k \in \mathbb{N}$, so $\sigma = (\sigma_0, \sigma_1, \ldots, \sigma_k, \ldots)$.

Causal functions on sequences

A^ω is the set of *A-valued infinite sequences*. The entries of $\sigma \in A^\omega$ are $[\sigma]_k = \sigma_k \in A$ for $k \in \mathbb{N}$, so $\sigma = (\sigma_0, \sigma_1, \ldots, \sigma_k, \ldots)$.

Slicing extracts a finite list from an infinite sequence:

$$[\cdot]_{i:j} : \sigma \mapsto (\sigma_i, \sigma_{i+1}, \ldots, \sigma_j)$$

Causal functions on sequences

A^ω is the set of *A-valued infinite sequences*. The entries of $\sigma \in A^\omega$ are $[\sigma]_k = \sigma_k \in A$ for $k \in \mathbb{N}$, so $\sigma = (\sigma_0, \sigma_1, \ldots, \sigma_k, \ldots)$.

Slicing extracts a finite list from an infinite sequence:

$$[\cdot]_{i:j} : \sigma \mapsto (\sigma_i, \sigma_{i+1}, \ldots, \sigma_j)$$

Definition

A *causal* function $f : A^\omega \to B^\omega$ satisfies

$$\forall \sigma, \tau \in A^\omega, \forall k \in \mathbb{N}, [\sigma]_{0:k} = [\tau]_{0:k} \to [f(\sigma)]_{0:k} = [f(\tau)]_{0:k}$$

Intuitively, the first k outputs of f only depend on the first k inputs.

Examples

1. $\bar{b} : 1^\omega \to B^\omega$ defined by $[\bar{b}]_k = b$ is causal.

Examples

1. $\bar{b} : 1^\omega \to B^\omega$ defined by $[\bar{b}]_k = b$ is causal.

2. If $g : A \to B$, then $\mathtt{map}(g) : A^\omega \to B^\omega$ defined by $[\mathtt{map}(g)(\sigma)]_k = g(\sigma_k)$ is causal.

Examples

1. $\bar{b} : 1^\omega \to B^\omega$ defined by $[\bar{b}]_k = b$ is causal.

2. If $g : A \to B$, then $\mathtt{map}(g) : A^\omega \to B^\omega$ defined by $[\mathtt{map}(g)(\sigma)]_k = g(\sigma_k)$ is causal.

3. $\mathtt{runProd} : \mathbb{R}^\omega \to \mathbb{R}^\omega$ defined by $[\mathtt{runProd}(\sigma)]_k = \prod_{i=0}^{k} \sigma_i$ is causal.

Examples

1. $\bar{b} : 1^\omega \to B^\omega$ defined by $[\bar{b}]_k = b$ is causal.

2. If $g : A \to B$, then $\mathtt{map}(g) : A^\omega \to B^\omega$ defined by $[\mathtt{map}(g)(\sigma)]_k = g(\sigma_k)$ is causal.

3. $\mathtt{runProd} : \mathbb{R}^\omega \to \mathbb{R}^\omega$ defined by $[\mathtt{runProd}(\sigma)]_k = \prod_{i=0}^{k} \sigma_i$ is causal.

4. $\mathtt{zip} : (A \times A)^\omega \to A^\omega$ defined by

$$[\mathtt{zip}(\sigma, \tau)]_k = \begin{cases} \sigma_{k/2} & \text{if } k \equiv 0 \pmod 2 \\ \tau_{(k-1)/2} & \text{if } k \equiv 1 \pmod 2 \end{cases}$$ is causal.

Examples

1. $\bar{b} : 1^\omega \to B^\omega$ defined by $[\bar{b}]_k = b$ is causal.

2. If $g : A \to B$, then $\mathtt{map}(g) : A^\omega \to B^\omega$ defined by $[\mathtt{map}(g)(\sigma)]_k = g(\sigma_k)$ is causal.

3. $\mathtt{runProd} : \mathbb{R}^\omega \to \mathbb{R}^\omega$ defined by $[\mathtt{runProd}(\sigma)]_k = \prod_{i=0}^{k} \sigma_i$ is causal.

4. $\mathtt{zip} : (A \times A)^\omega \to A^\omega$ defined by

$$[\mathtt{zip}(\sigma, \tau)]_k = \begin{cases} \sigma_{k/2} & \text{if } k \equiv 0 \pmod 2 \\ \tau_{(k-1)/2} & \text{if } k \equiv 1 \pmod 2 \end{cases}$$ is causal.

5. $\mathtt{tl} : A^\omega \to A^\omega$ defined by $[\mathtt{tl}(\sigma)]_k = \sigma_{k+1}$ is **not causal**, since output k is input $k + 1$.

Background

Causal functions appear all over computer science:

1. Mealy machines [Mealy, 1955 and Eilenberg, 1974]
2. Coalgebra [Rutten, 2006]
3. Synchronous digital circuits [Leiserson and Saxe, 1986]
4. Signal flow graphs [Bonchi, Sobociński and Zanasi, 2014 and Milius, 2010]
5. Recurrent neural networks [Rumelhart, 1986 and Hochreiter and Schmidhuber, 1997]

(If you have examples from mathematics where this class of functions is important, let me know!)

Automata theory / coalgebraic approach

Mealy machines with input alphabet A and output alphabet B have behaviors much like causal functions $A^\omega \to B^\omega$.

Automata theory / coalgebraic approach

Mealy machines with input alphabet A and output alphabet B have behaviors much like causal functions $A^\omega \to B^\omega$.

Indeed, causal functions $A^\omega \to B^\omega$ carry a final coalgebra for the functor $M_{A,B}(X) = (B \times X)^A$. [Rutten, 2006]

Automata theory / coalgebraic approach

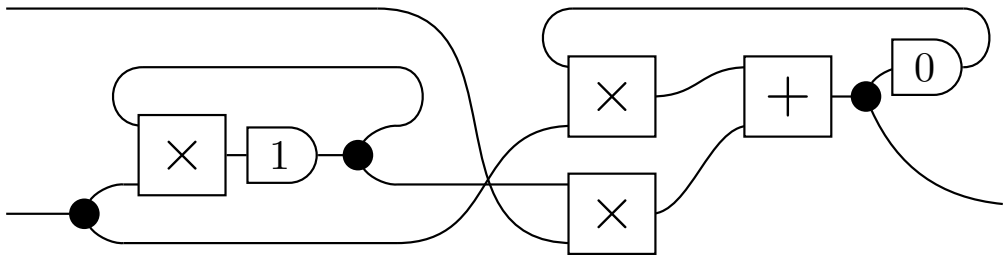
Mealy machines with input alphabet A and output alphabet B have behaviors much like causal functions $A^\omega \to B^\omega$.

Indeed, causal functions $A^\omega \to B^\omega$ carry a final coalgebra for the functor $M_{A,B}(X) = (B \times X)^A$. [Rutten, 2006]

Thus one way to specify a causal function is to specify a state in a Mealy machine. This is the basis of the *syntactic (GSOS) method* in stream differential equations. [Hansen, Kupke and Rutten, 2017]

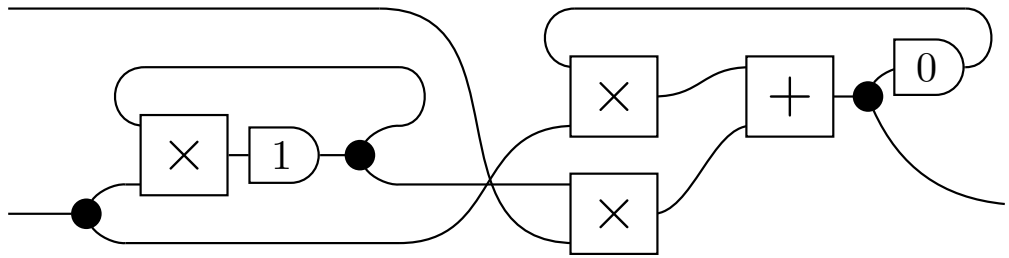
Digital circuit / signal flow graph approach

Circuit diagram treatments of causal functions are more explicit about how state is stored, how it is updated, and how outputs are produced. For example:

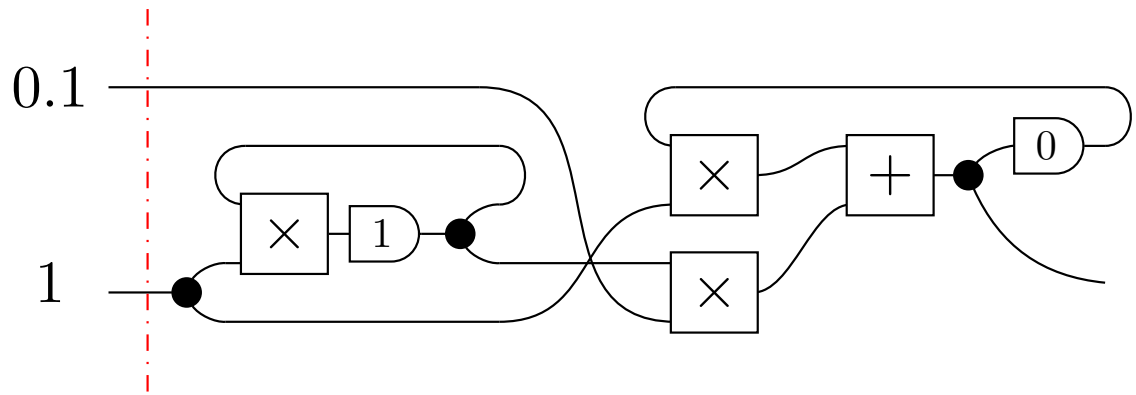


Digital circuit / signal flow graph approach

Circuit diagram treatments of causal functions are more explicit about how state is stored, how it is updated, and how outputs are produced. For example:

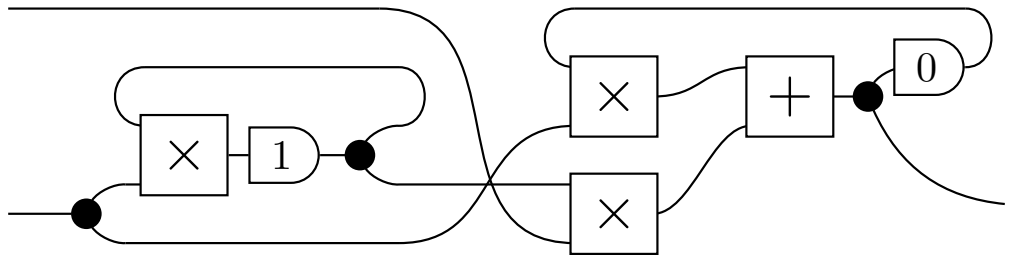


Let's evaluate this diagram once:

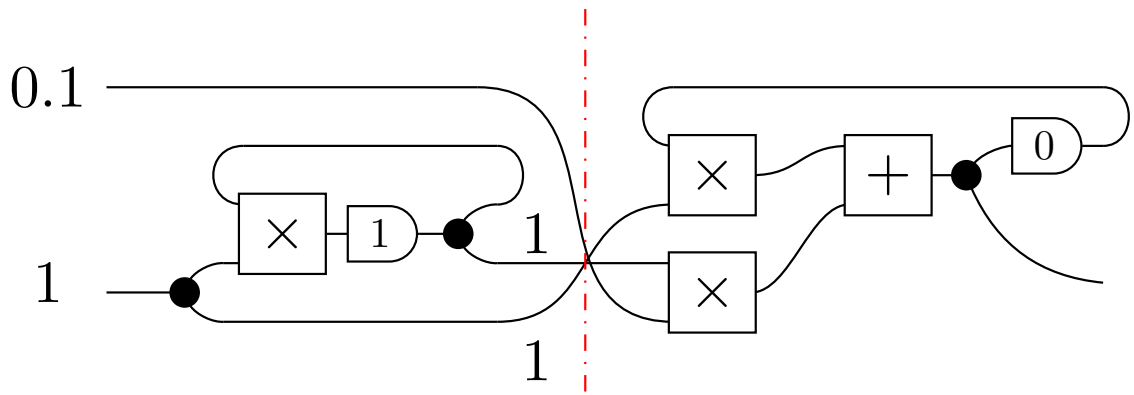


Digital circuit / signal flow graph approach

Circuit diagram treatments of causal functions are more explicit about how state is stored, how it is updated, and how outputs are produced. For example:

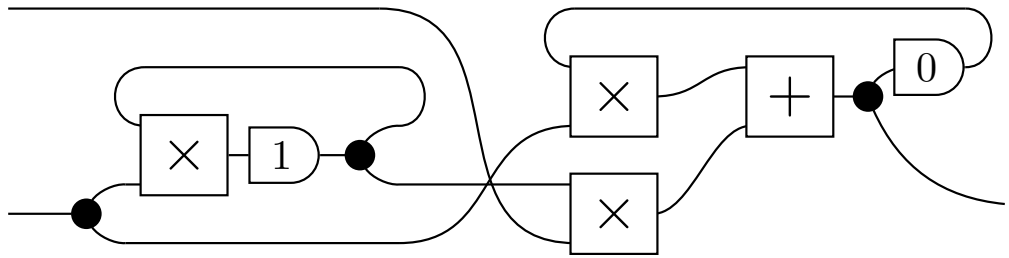


Let's evaluate this diagram once:

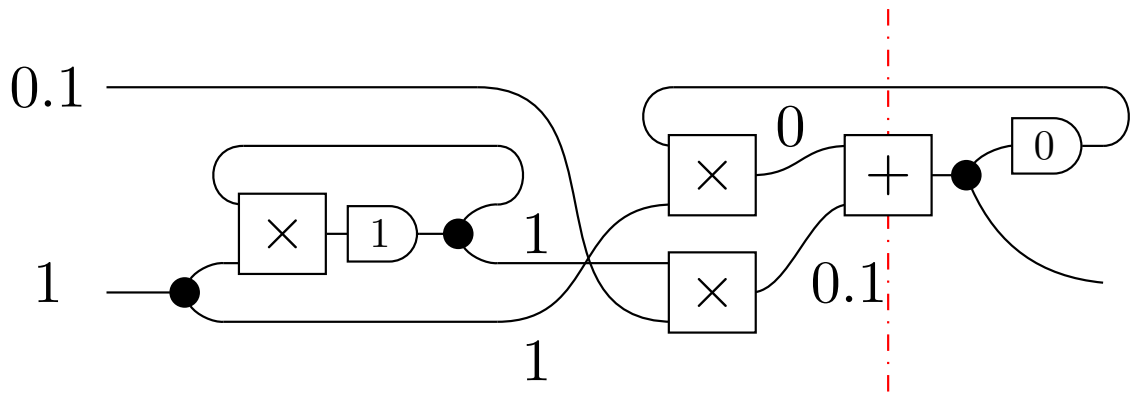


Digital circuit / signal flow graph approach

Circuit diagram treatments of causal functions are more explicit about how state is stored, how it is updated, and how outputs are produced. For example:

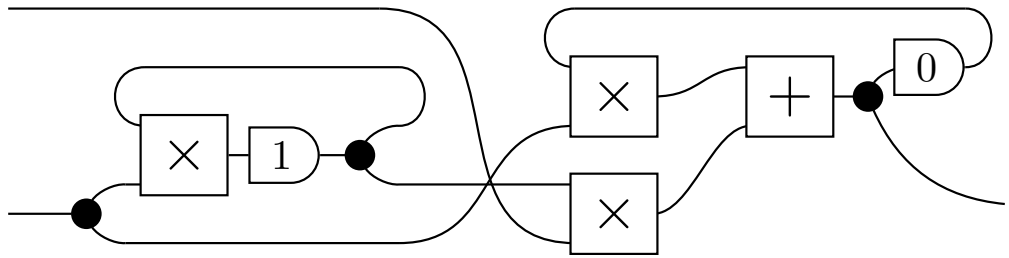


Let's evaluate this diagram once:

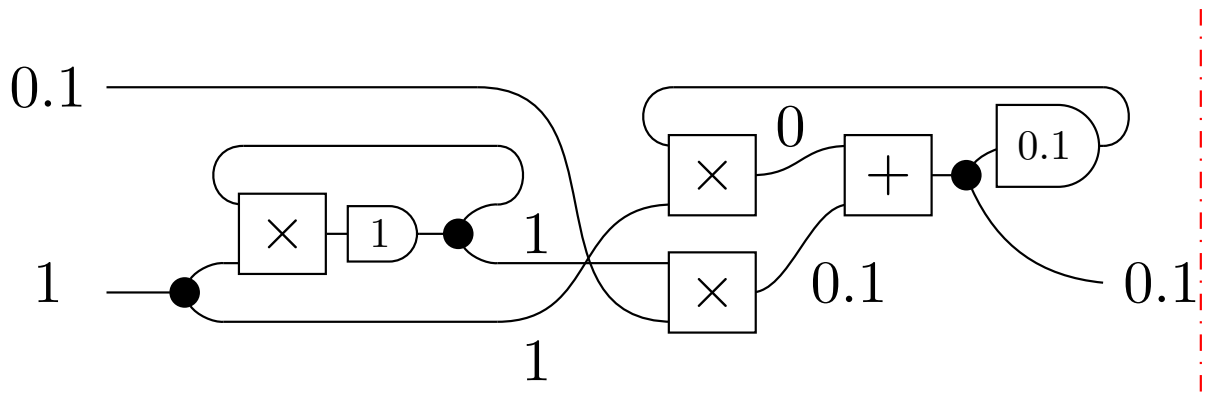


Digital circuit / signal flow graph approach

Circuit diagram treatments of causal functions are more explicit about how state is stored, how it is updated, and how outputs are produced. For example:



Let's evaluate this diagram once:



Recurrent neural networks

RNNs are special instances of the circuit diagram approach with the requirement that the functions involved are *differentiable*.

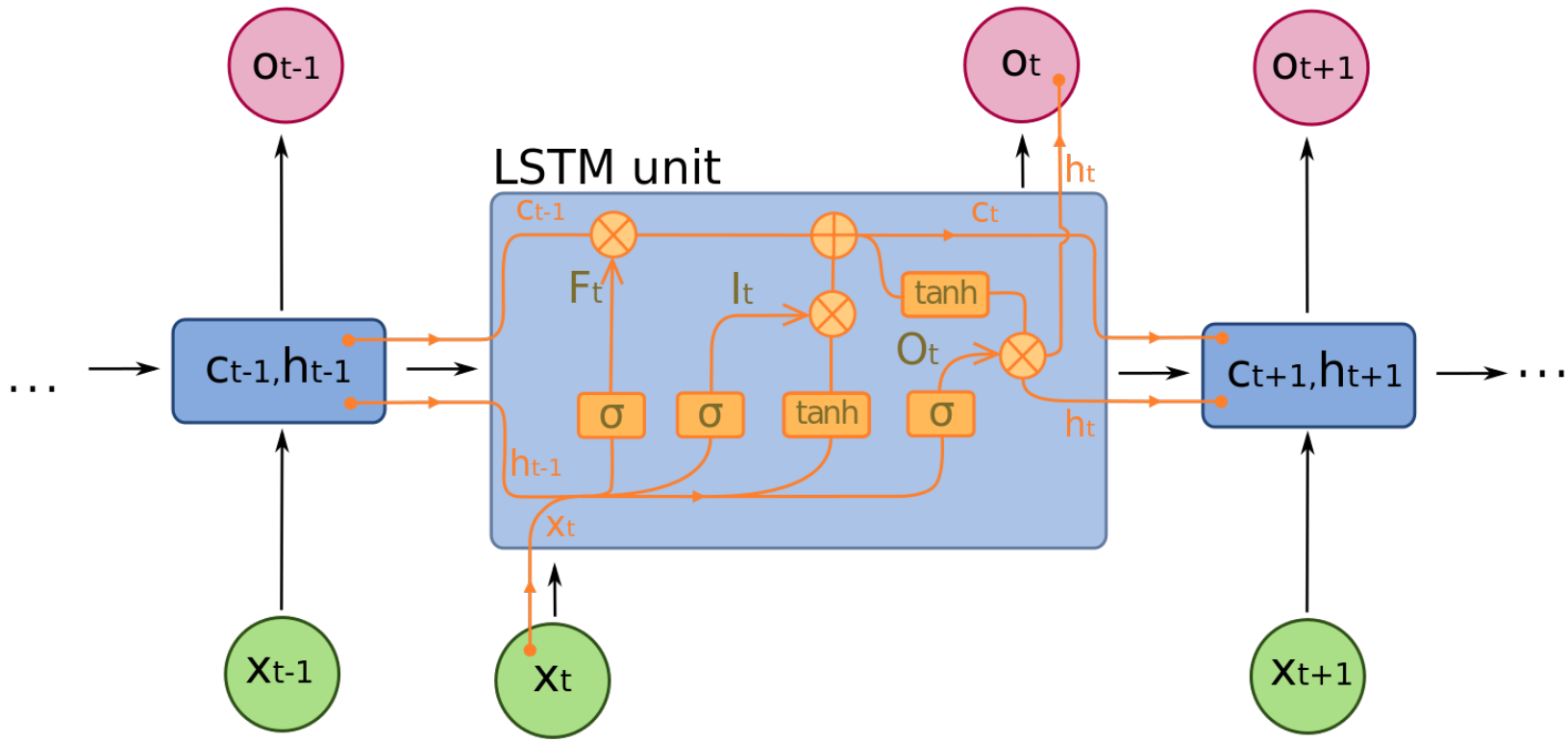


Figure: François Deloche - Own work, CC BY-SA 4.0, wikimedia

Finite approximations approach

Another perspective on causal functions is to treat them as a limit of functions on finite lists. This is based on the following observation:

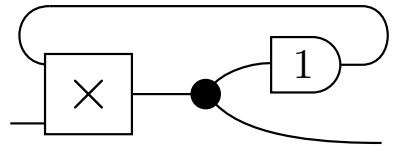
Theorem

The following are equivalent:

1. *a causal function* $f : A^\omega \to B^\omega$,
2. *an infinite sequence of functions* $u_k : A^{k+1} \to B$, *and*
3. *an infinite sequence of functions* $t_k : A^{k+1} \to B^{k+1}$ *satisfying* $[t_{k+1}(\vec{a})]_{0:k} = t_k([\vec{a}]_{0:k})$ *for all* $\vec{a} \in A^{k+2}$.

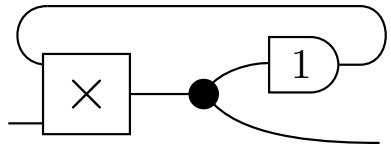
Proof Idea. u_k gives the $k+1^{\text{st}}$ entry in the sequence produced by f, and t_k provides the first $k+1$ entries.

What causal function is represented by this circuit?

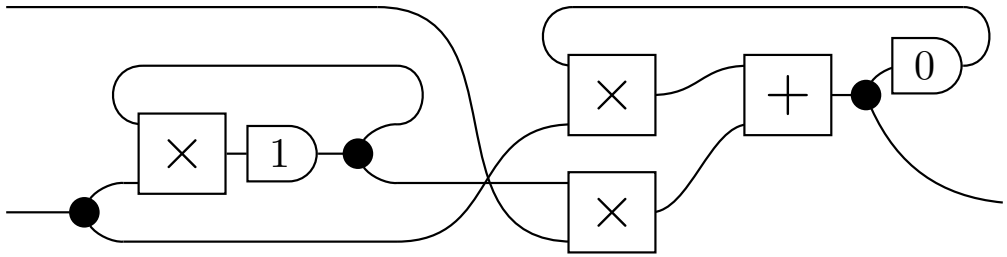


Where we're heading, update I

I would like to convince you the derivative of the running product:



Is this:



Outline

Main ideas

We start from a (strictified Cartesian) category $\mathbb{C}$ whose morphisms represent simple computations. Our goal is to define an extended category whose morphisms are causal functions.

Main ideas

We start from a (strictified Cartesian) category $\mathbb{C}$ whose morphisms represent simple computations. Our goal is to define an extended category whose morphisms are causal functions.

There are three steps:

1. describe a single step of the computation, including a mechanism for sending and receiving state
2. chain single steps together to get a computation sequence
3. quotient these computation sequences by their observable behaviour.

Step 1: Single computation steps

We separate inputs and outputs of a $\mathbb{C}$-morphism into two classes: *values*, exchanged with the environment, and *states*, exchanged with next/previous steps of the computation.

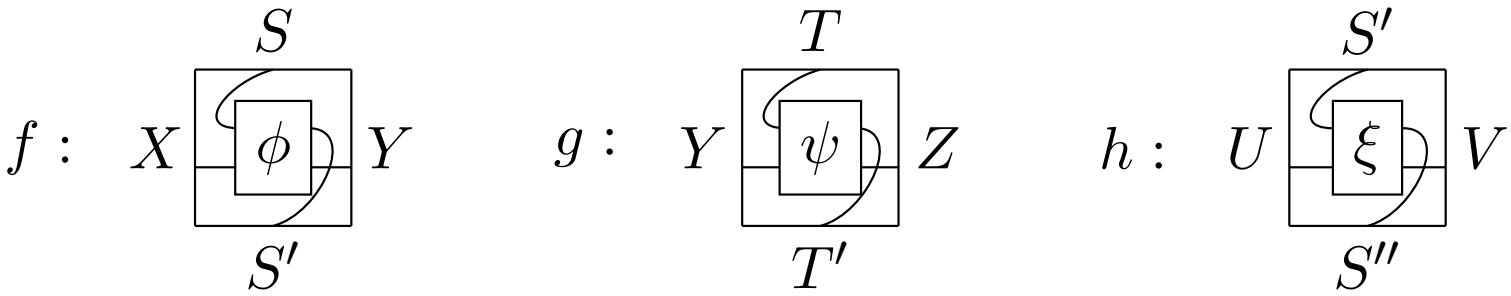
Definition

A *computation step* $f : X \xrightarrow[S']{S} Y$ is four objects and a morphism:

1. S — the input state received from the previous step
2. X — the input value received from the environment
3. S' — the output state sent to the next step
4. Y — the output value sent to the environment
5. $\phi : S \times X \to S' \times Y$ — the $\mathbb{C}$-morphism doing the computation

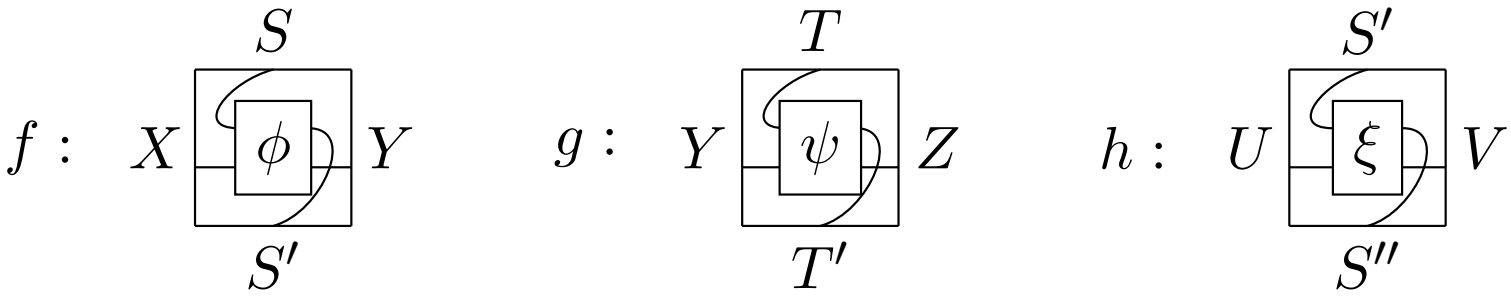
Step 1: Single computation steps

We can draw a single step like so:

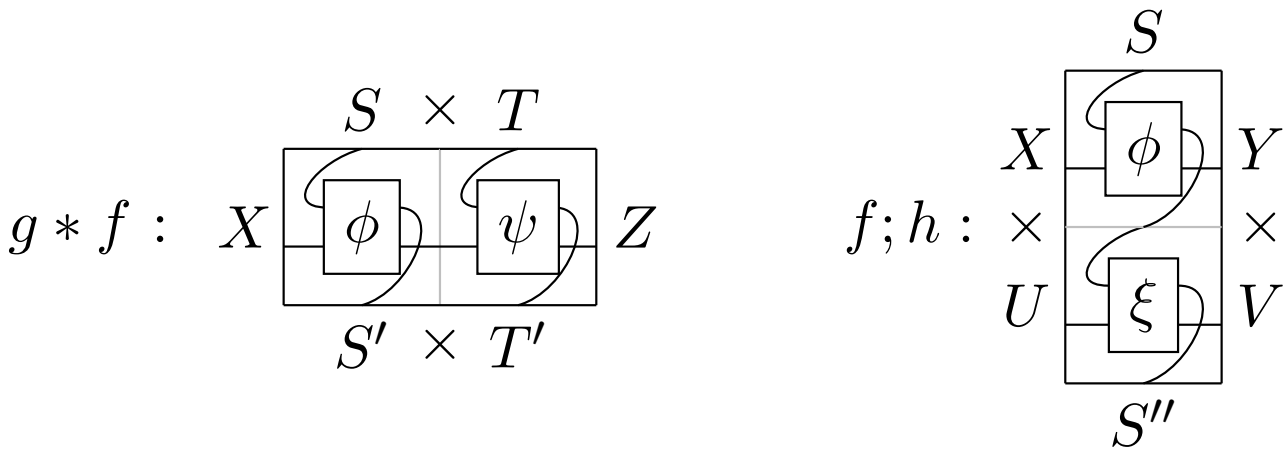


Step 1: Single computation steps

We can draw a single step like so:

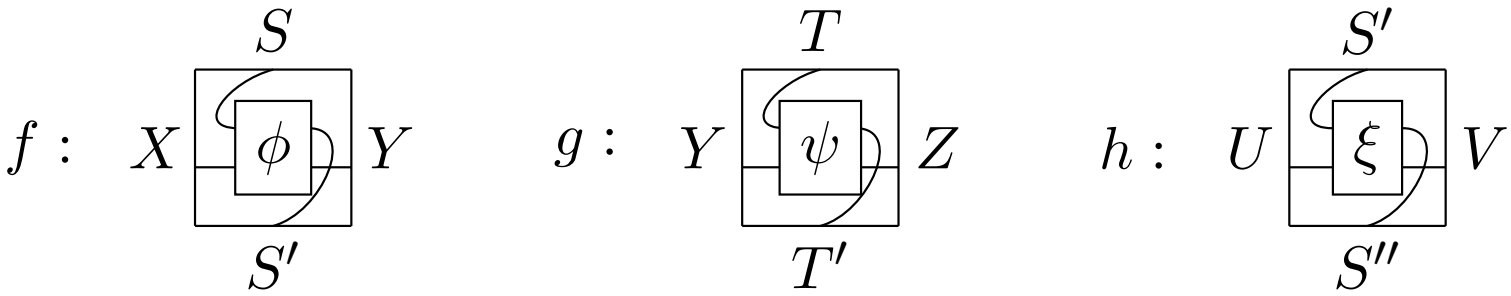


Computation steps can be composed in two different ways:

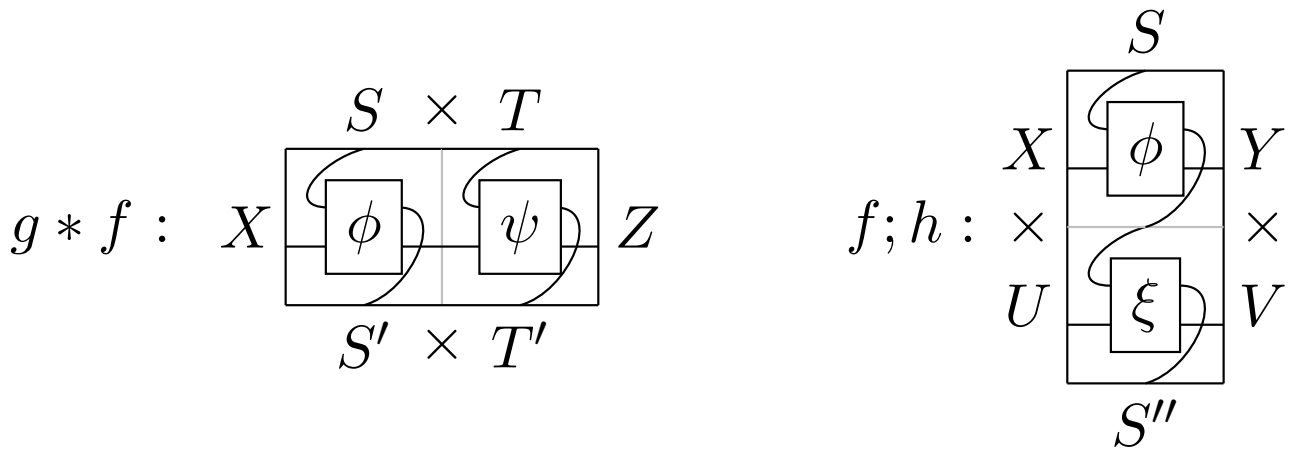


Step 1: Single computation steps

We can draw a single step like so:



Computation steps can be composed in two different ways:



psst . . . double category

Step 2: Computation sequences

Next, we chain together infinitely many of these computation steps to compute a sequence of outputs.

Definition

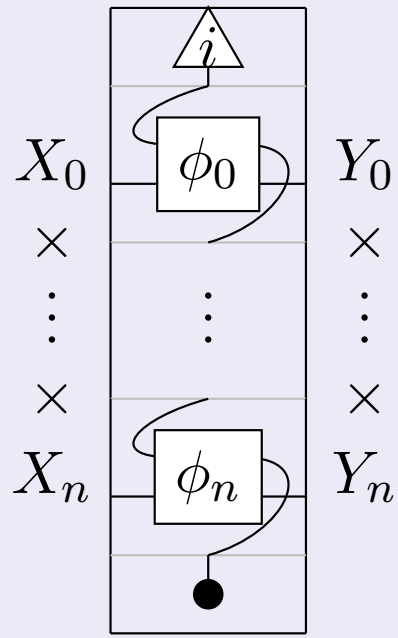
A *computation sequence* $\mathbf{f} = (i, [f_k])$ is an infinite sequence of computation steps $f_k : X_k \xrightarrow[S_{k+1}]{S_k} Y_k$, and an initial state $i : 1 \xrightarrow[S_0]{1} 1$. We say $\mathbf{f}$ takes sequences of type $[X_k] = (X_0, X_1, \ldots)$ to sequences of type $[Y_k] = (Y_0, Y_1, \ldots)$.

Note that f_k and f_{k+1} are vertically composable, as are i and f_0.

Step 3: Comparing computation sequences

Definition

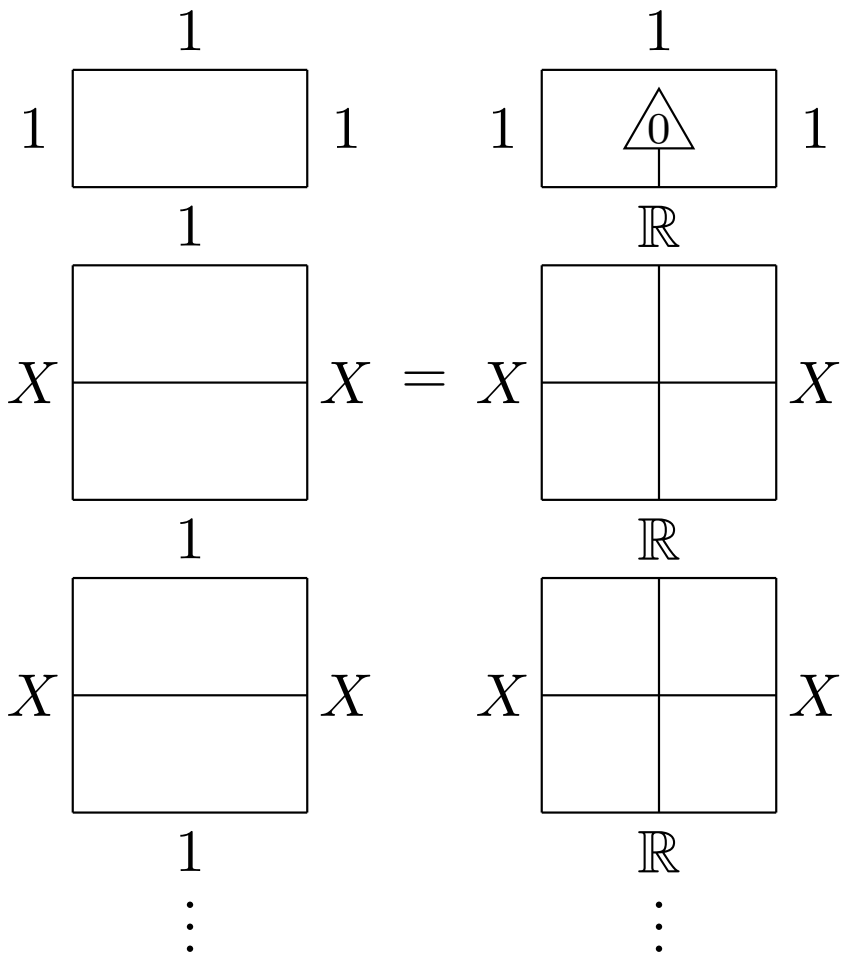
The n*th truncation* of a computation sequence is the morphism of the vertical composite of the first $n+1$ steps:



Definition

Two computation sequences $\mathbf{f}, \mathbf{g} : [X_k] \to [Y_k]$ are *extensionally equivalent* means they have the same nth truncation for all $n \in \mathbb{N}$.

Step 3: Extensionally equivalent identity computations



Step 3: Comparing computation sequences

Definition

A *stateful (sequence) function* is an extensional equivalence class of computation sequences.

Definition

If $\mathbb{C}$ is a (strict) Cartesian category, then its *stateful extension* is a category $\mathrm{St}(\mathbb{C})$ where

- objects are $[X_k]$, i.e. infinite sequences of objects in $\mathbb{C}$ and
- morphisms are stateful functions $\mathbf{f} : [X_k] \to [Y_k]$.

Sanity check

Theorem

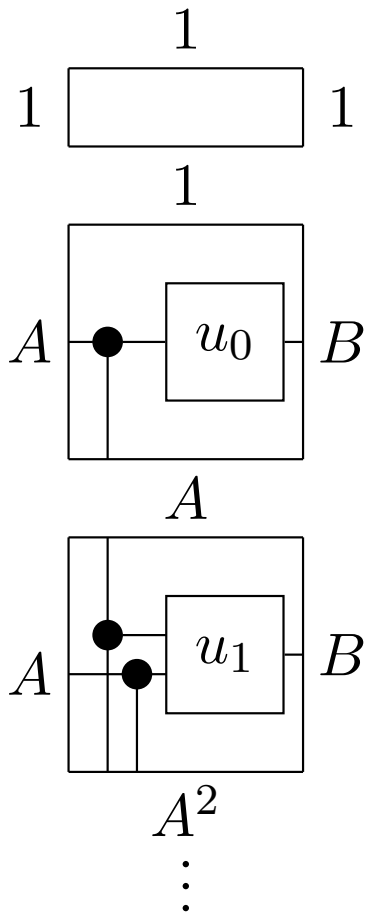
The homset $\mathrm{St}(\mathit{Set})([A],[B])$ *is in 1-1 correspondence with the set of causal functions from* A^ω *to* B^ω*.*

Sanity check

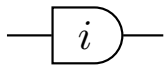
Theorem

The homset $\mathrm{St}(Set)([A],[B])$ *is in 1-1 correspondence with the set of causal functions from* A^ω *to* B^ω*.*

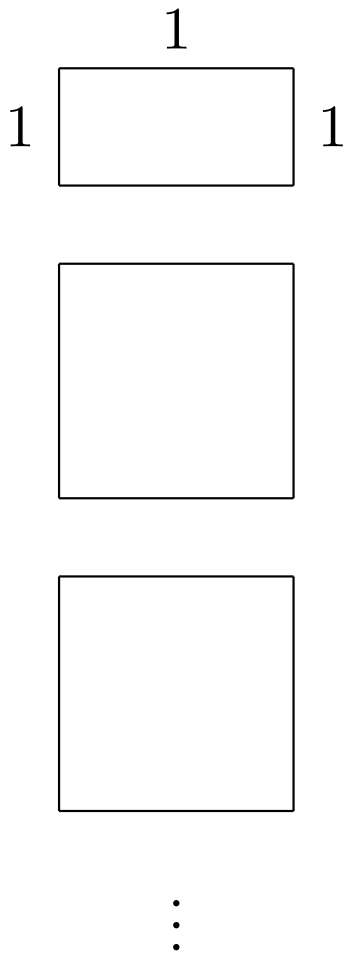
Idea. A causal $f : A^\omega \to B^\omega$ determines a unique sequence of functions $u_k : A^{k+1} \to B$. Form a computation sequence as follows:



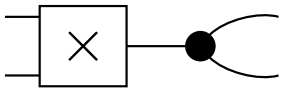
Quiz

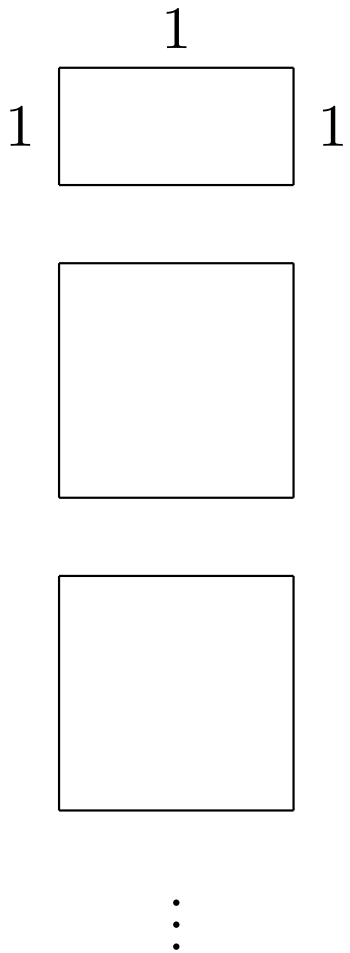
How would you implement 

as a computation sequence?

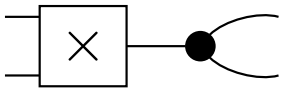


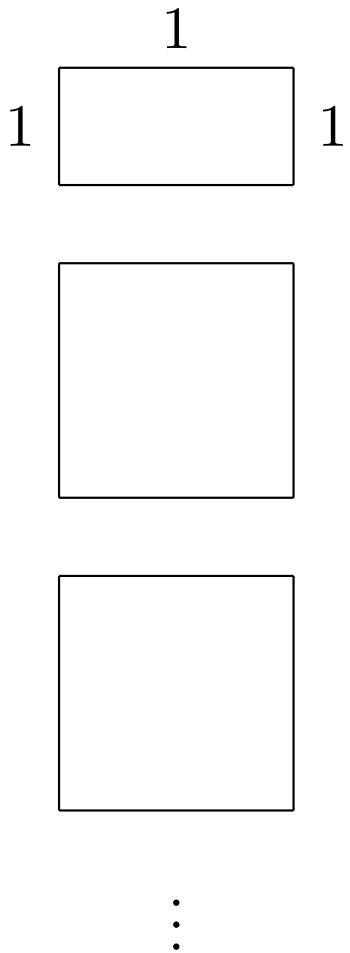
Quiz

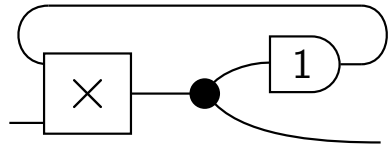
How would you implement  as a computation sequence?

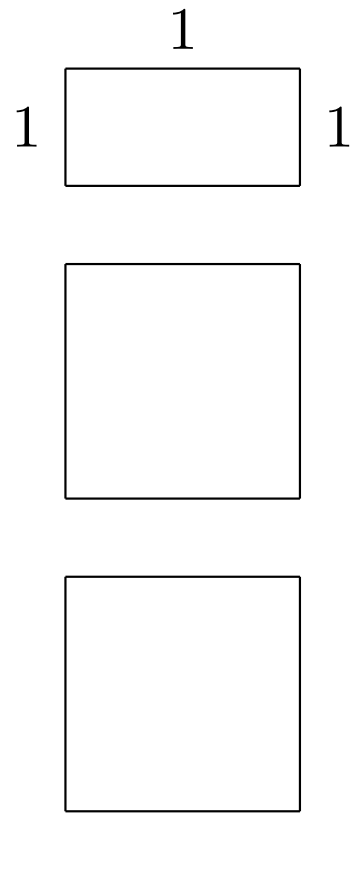


Quiz

How would you implement  as a computation sequence?



How would you implement  as a computation sequence?

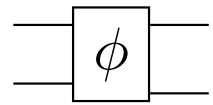


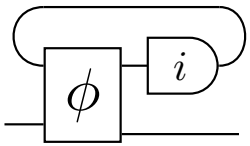
Outline

Delayed trace

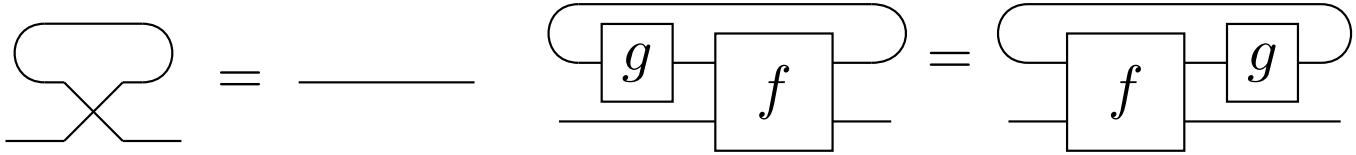
This loop-with-delay-gate is a trace-like operation.

$$\frac{\phi : S \times X \to S \times Y}{dtr_i^S(\phi) : X \to Y}$$

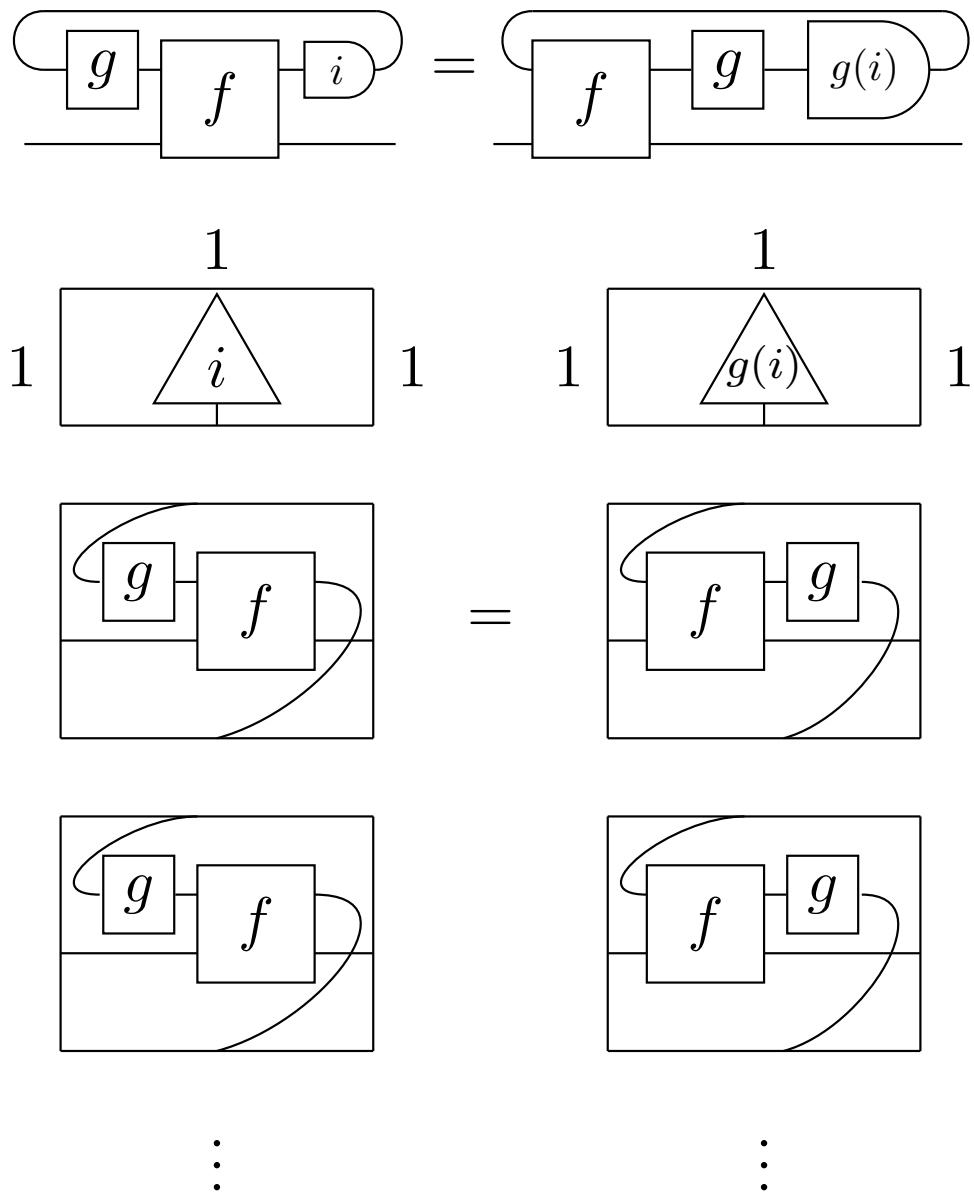




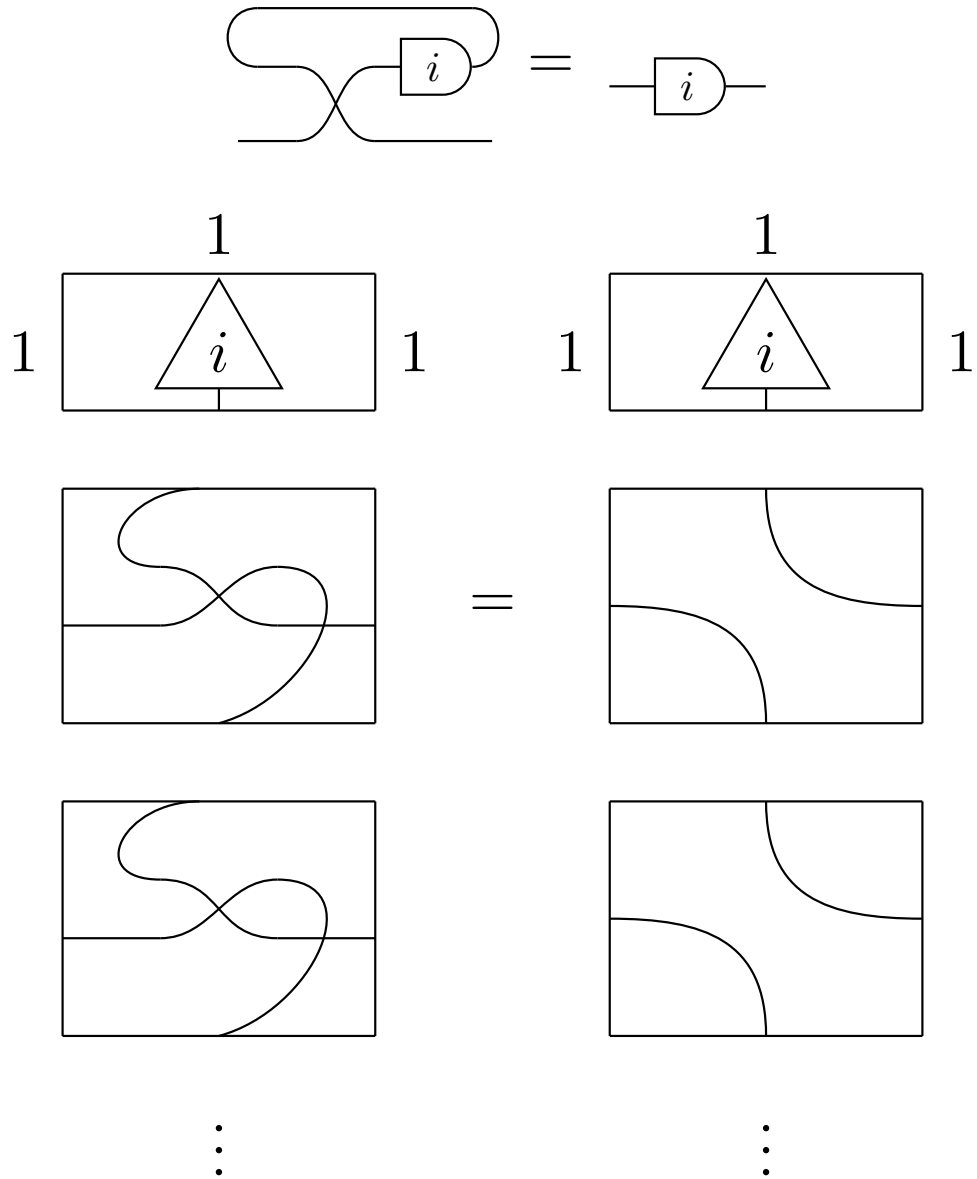
It satisfies **most** of the the trace axioms [Joyal, Street, Verity, 1996] but misses two: yanking and dinaturality. For monoidal trace, those are



~~Dinaturality~~ → retiming



~~Yanking~~ → delay



Advantages of delayed trace

Often, people formalizing circuits in category theory use these steps:

1. add registers to the category,
2. add trace to the category,
3. restrict to a subcategory such that all traces are taken on positions guarded by a register.

Advantages of delayed trace

Often, people formalizing circuits in category theory use these steps:

1. add registers to the category,
2. add trace to the category,
3. restrict to a subcategory such that all traces are taken on positions guarded by a register.

Our idea is to:

1. add a delayed trace to the category,
2. recover registers by delay-tracing symmetry

This ensures all loops are guarded (without a syntactic restriction!) and requires less structure.

Advantages of delayed trace

Often, people formalizing circuits in category theory use these steps:

1. add registers to the category,
2. add trace to the category,
3. restrict to a subcategory such that all traces are taken on positions guarded by a register.

Our idea is to:

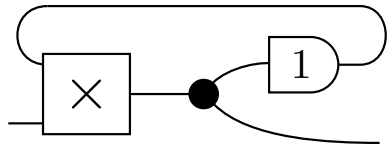
1. add a delayed trace to the category,
2. recover registers by delay-tracing symmetry

This ensures all loops are guarded (without a syntactic restriction!) and requires less structure.

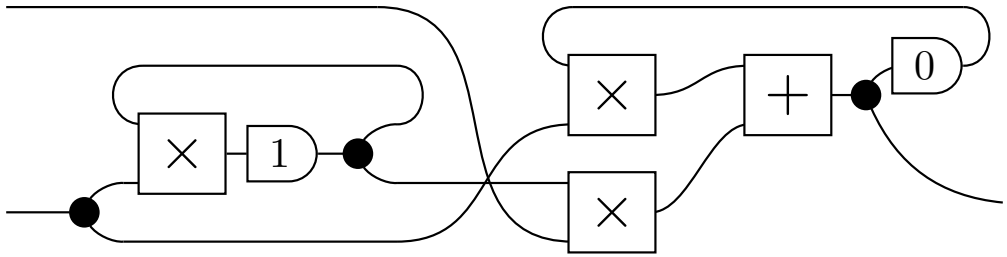
We are also interested in looking at delayed traces abstractly...

Where we're heading, update II

I would like to convince you the derivative of the running product:



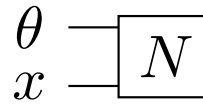
Is this:



Now we know how to represent these circuits as computation sequences.

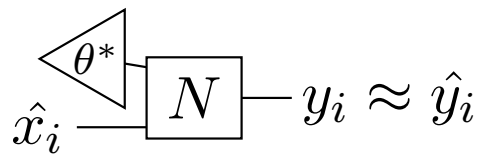
Outline

(Feedforward) neural networks

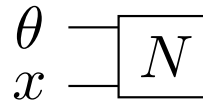
A *neural network* looks like 

. We assume its input and output spaces are real Euclidean spaces $\mathbb{R}^n$.

Training a neural network (with data $\hat{x}_i, \hat{y}_i$) means finding θ^* so that:

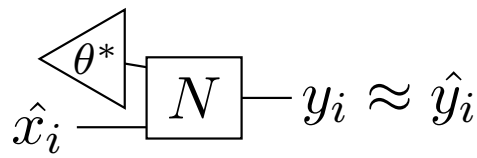


(Feedforward) neural networks

A *neural network* looks like 

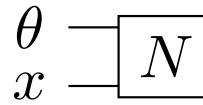
. We assume its input and output spaces are real Euclidean spaces $\mathbb{R}^n$.

Training a neural network (with data $\hat{x}_i, \hat{y}_i$) means finding θ^* so that:



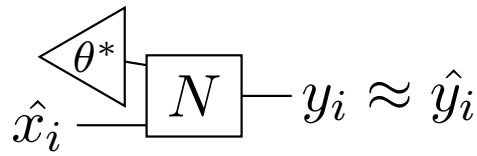
Gradient-based training algorithms are based on the insight that $\frac{\partial N}{\partial \theta}$ is a good approximation for the change in y that results from a small change in θ. This allows us to make smart updates to θ^*.

(Feedforward) neural networks

A *neural network* looks like 

. We assume its input and output spaces are real Euclidean spaces $\mathbb{R}^n$.

Training a neural network (with data $\hat{x}_i, \hat{y}_i$) means finding θ^* so that:

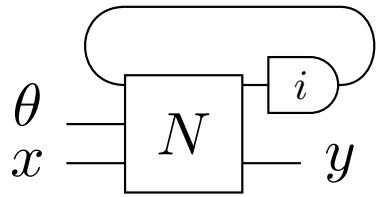


Gradient-based training algorithms are based on the insight that $\frac{\partial N}{\partial \theta}$ is a good approximation for the change in y that results from a small change in θ. This allows us to make smart updates to θ^*.

Various algorithms can be used to compute these partial derivatives, including *backpropagation* [Rumelhart, 1986].

(Feedback/recurrent) neural networks

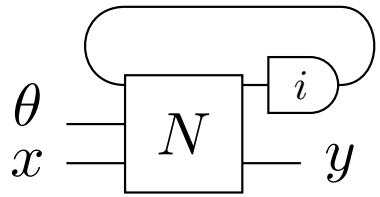
Recurrent neural networks (RNNs) are designed to process sequences of inputs using some internal state:



It's less clear what the derivatives of these functions are.

(Feedback/recurrent) neural networks

Recurrent neural networks (RNNs) are designed to process sequences of inputs using some internal state:

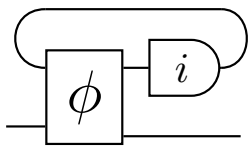


It's less clear what the derivatives of these functions are.

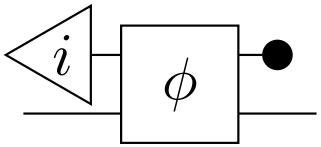
The most common training method for RNNs relies on a gradient-finding algorithm called *backpropagation through time*.

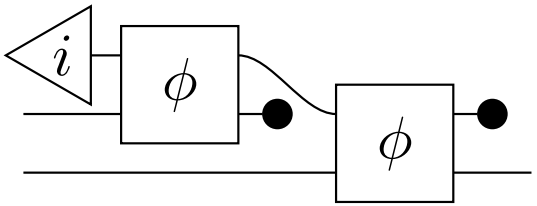
Backpropagation through time

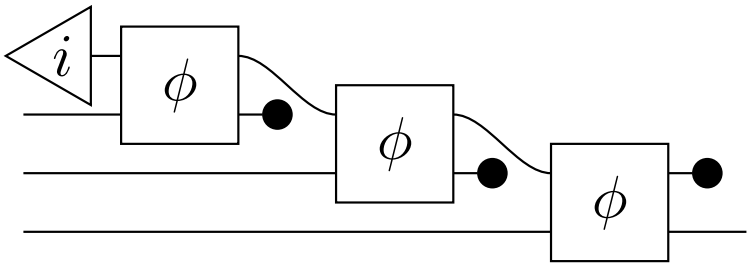
First, we *unroll* the network:

$\Phi =$ 

$\Rightarrow$

$u_0 =$ 

$u_1 =$ 

$u_2 =$ 

$\vdots$

Then, whenever the derivative of Φ is needed at an input of length $k+1$, the derivative of u_k is supplied instead.

Does BPTT make sense, or is it just a hack?

Does BPTT make sense, or is it just a hack?

"Making sense" means having the usual properties of derivatives:

1. a sum rule,
2. a chain rule,
3. being linear when evaluated at any base point,
4. symmetry of mixed partial derivatives,

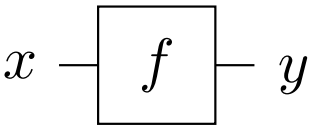
⋮

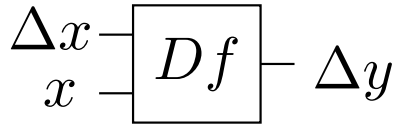
Outline

Derivatives, diagrammatically

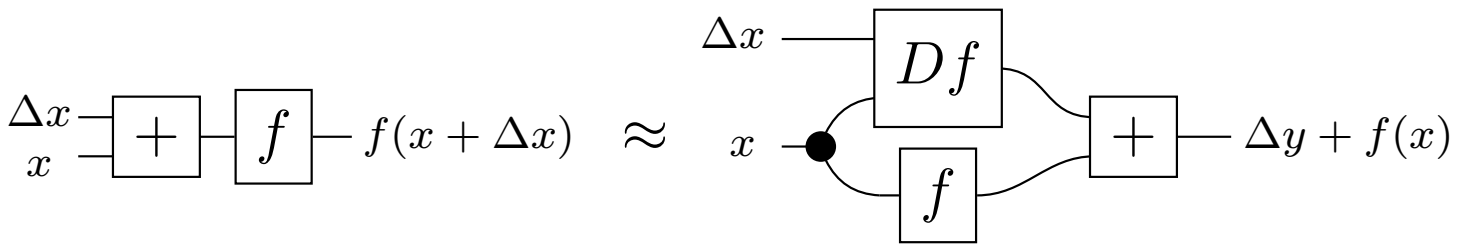
The derivative of x — f — y is $\begin{matrix}\Delta x \\ x\end{matrix}$ — Df — Δy, which gives the best linear approximation to f at a base point x.

Derivatives, diagrammatically

The derivative of 

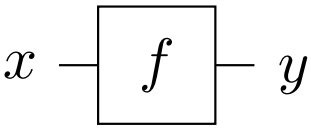
is 

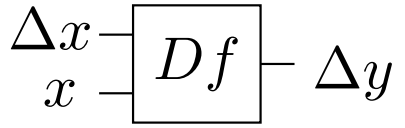
, which gives the best linear approximation to f at a base point x. That is



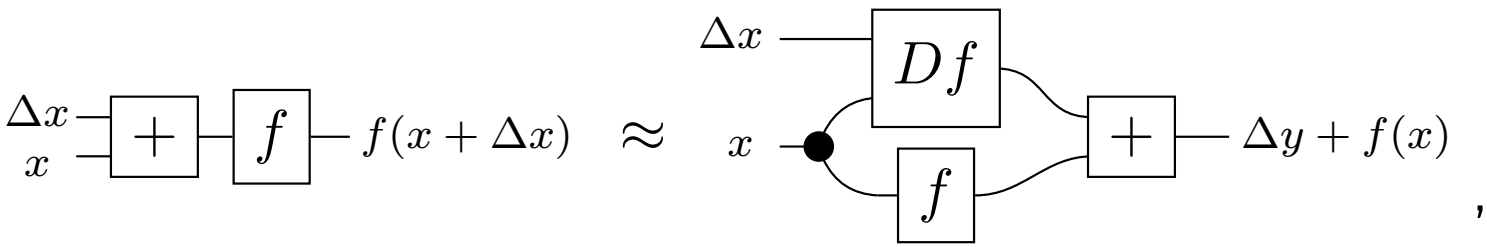
,

Derivatives, diagrammatically

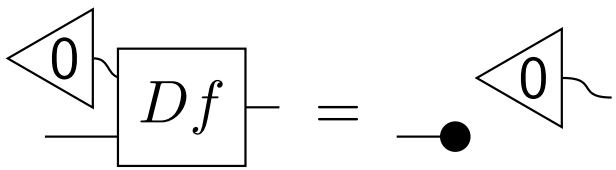
The derivative of 

is 

, which gives the best linear approximation to f at a base point x. That is

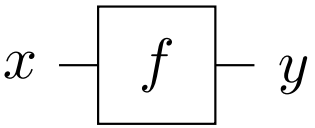


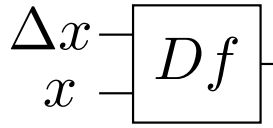
,

(CD2) 

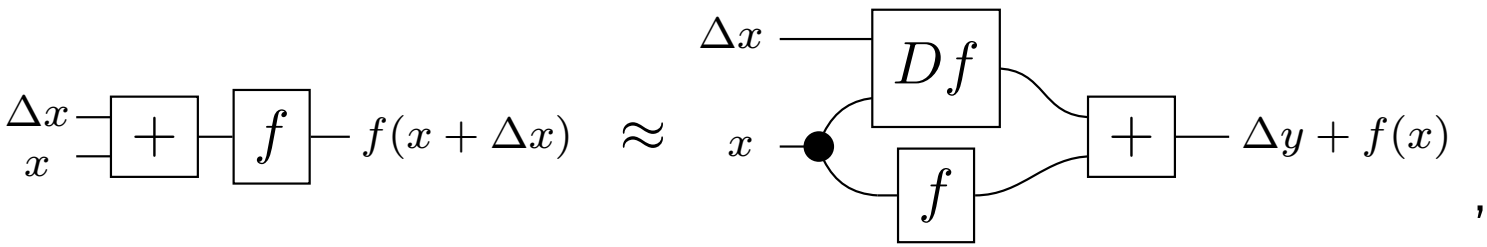
, and

Derivatives, diagrammatically

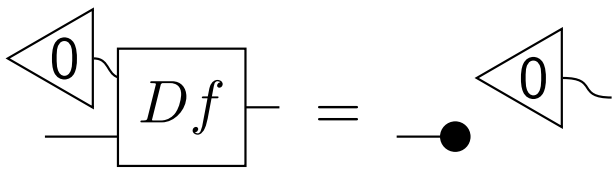
The derivative of 

is 

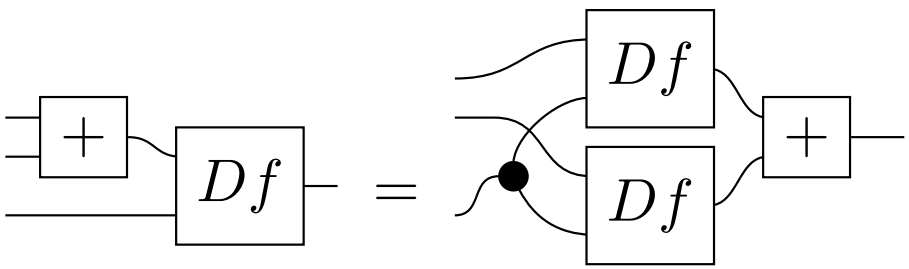
, which gives the best linear approximation to f at a base point x. That is



,

(CD2) 

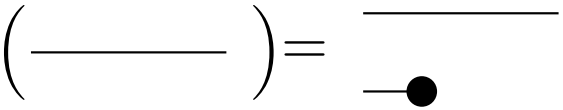
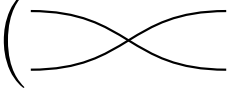
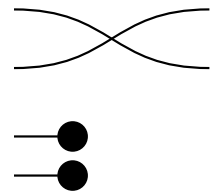
, and

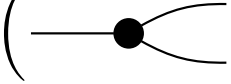
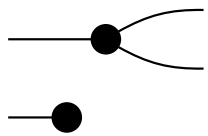
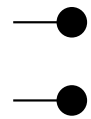
(CD3) 

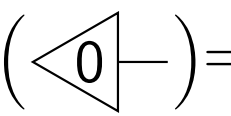
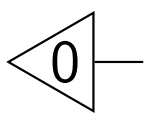
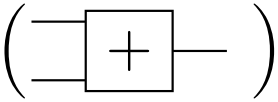
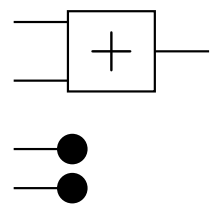
.

Derivatives of basic functions

The derivatives of some basic functions include:

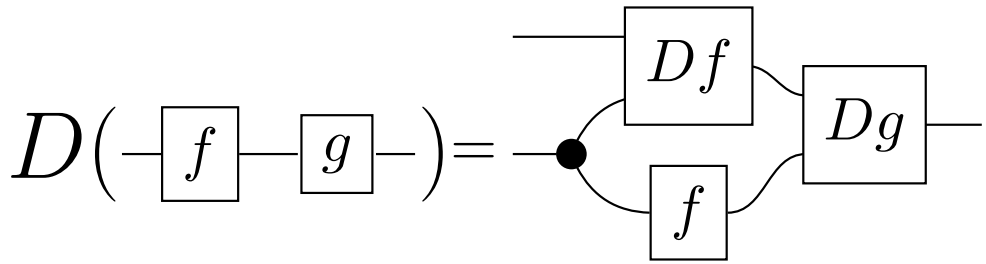
(CD1) $D($  $)=$ $D($  $)=$ 

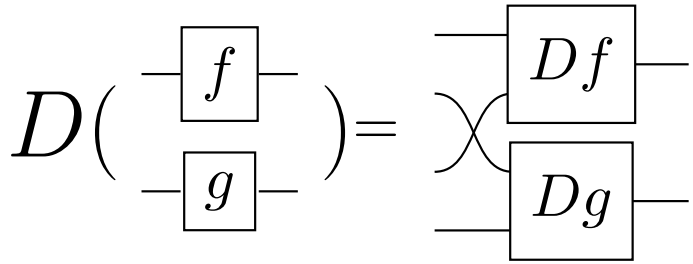
$D($  $)=$  $D($ $)=$ 

$D($  $)=$  $D($  $)=$ 

Properties of derivatives, diagrammatically

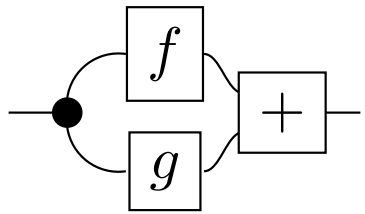
We can also express the chain rule and "parallel rule":

(CD4) 

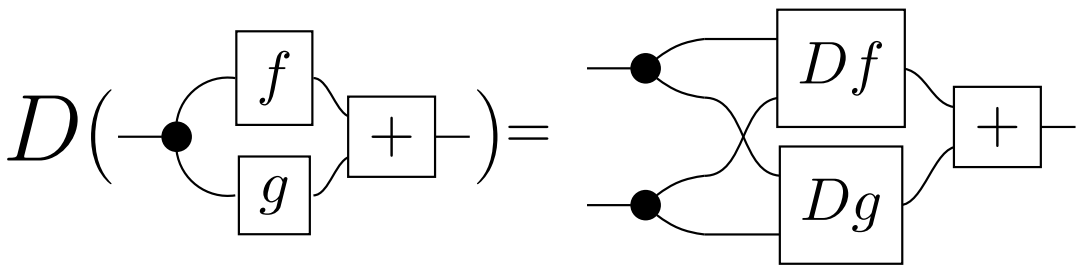
(CD5) 

Sum rule, diagrammatically

The sum rule is a consequence of the axioms we have seen so far. Here's a diagram for the sum of two functions:



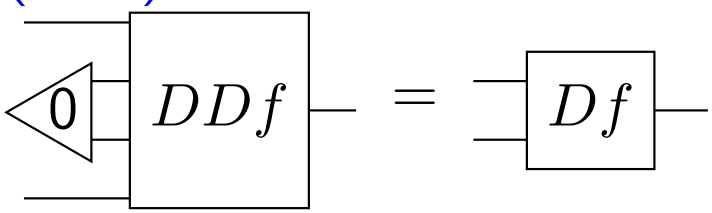
The rules we've seen allows us to compute:



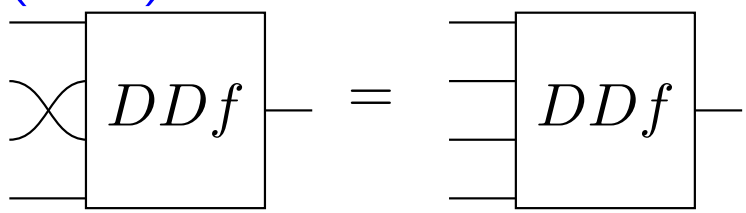
Properties of higher-order derivatives

It's also possible to express properties of higher order derivatives, but it's not worth the time to explain them in much detail.

(CD6)



(CD7)



Cartesian differential categories [Blute, Cockett, Seely '09]

A *Cartesian differential category* is a

- Cartesian category
- with left additive structure (providing 0 and $+$)
- an operation sending $f : X \to Y$ to $Df : X \times X \to Y$
- which satisfies CD1-7.

This is a categorical representation of the major algebraic properties of derivatives.

Cartesian differential categories [Blute, Cockett, Seely '09]

A *Cartesian differential category* is a

- Cartesian category
- with left additive structure (providing 0 and $+$)
- an operation sending $f : X \to Y$ to $Df : X \times X \to Y$
- which satisfies CD1-7.

This is a categorical representation of the major algebraic properties of derivatives.

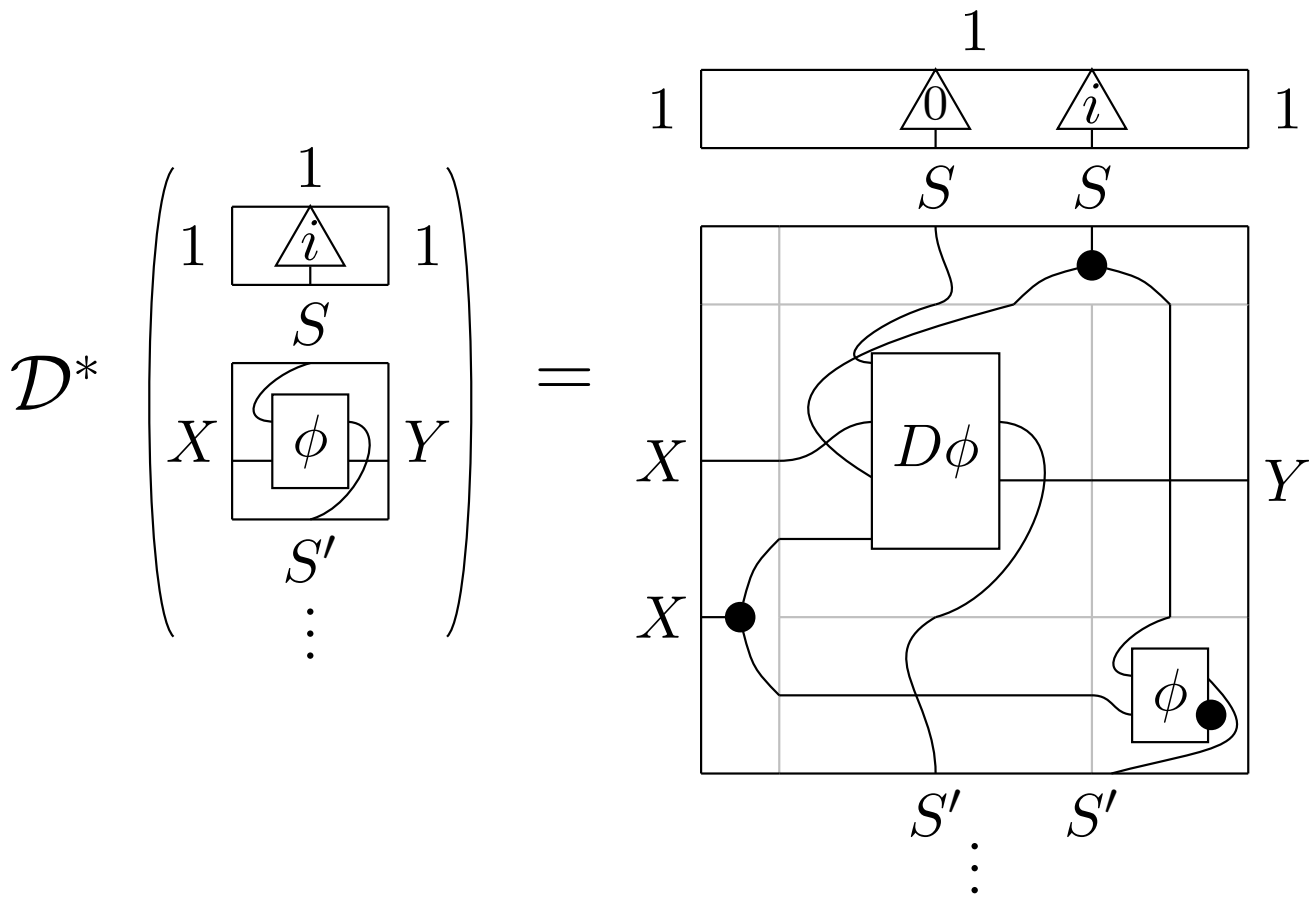
The canonical example of a Cartesian differential category is $\mathbf{Euc}_\infty$, where objects are $\mathbb{R}^n$ for all $n \in \mathbb{N}$ and morphisms are smooth functions.

Key contribution: differential operator lifts

Theorem (Katsumata, S)

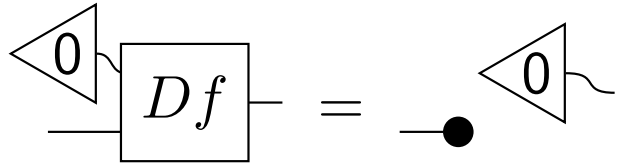
If $\mathbb{C}$ is a Cartesian differential category, so is $\mathrm{St}(\mathbb{C})$.

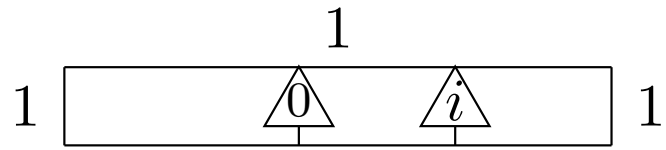
The Cartesian differential operator on $\mathrm{St}(\mathbb{C})$ is defined as follows:

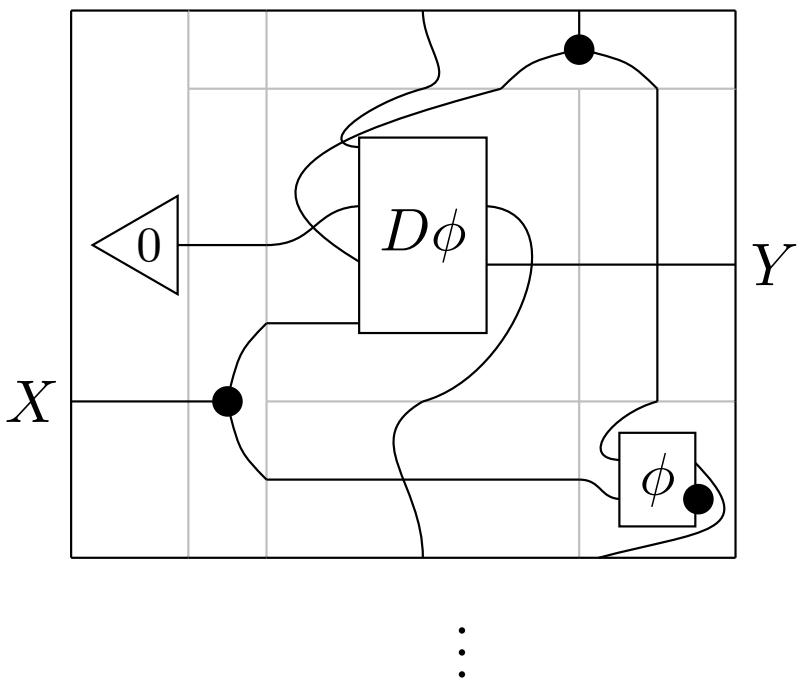


Key contribution: differential operator lifts

Proof idea. For CD2:

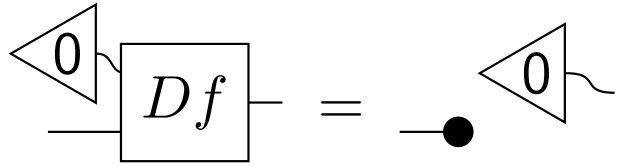


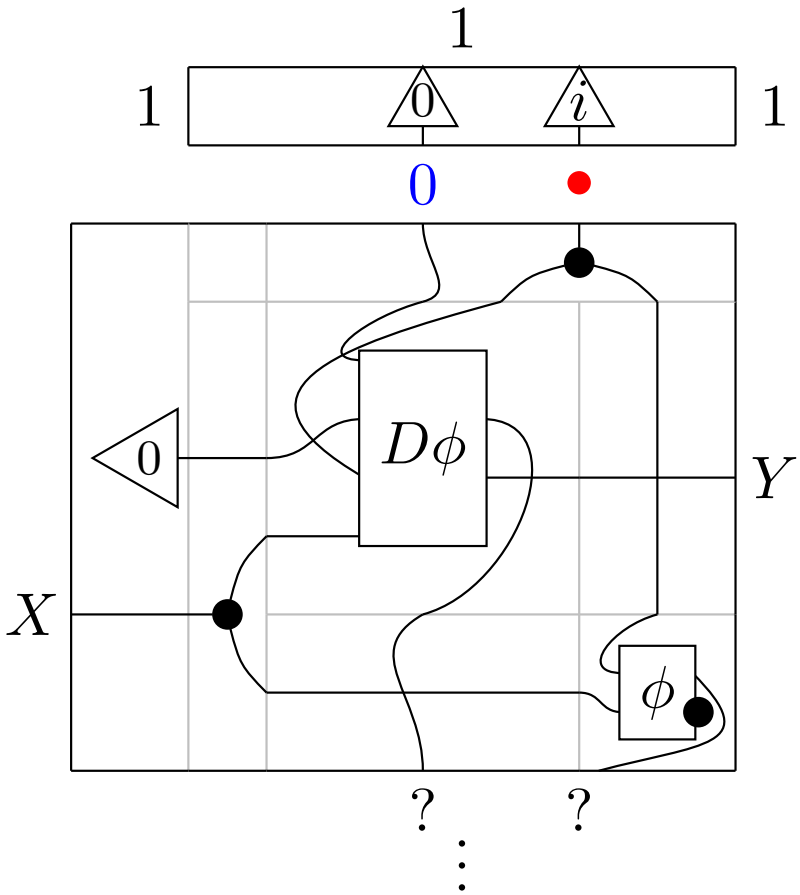




Key contribution: differential operator lifts

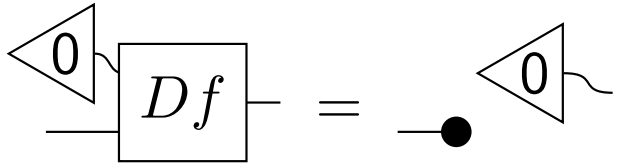
Proof idea. For CD2:

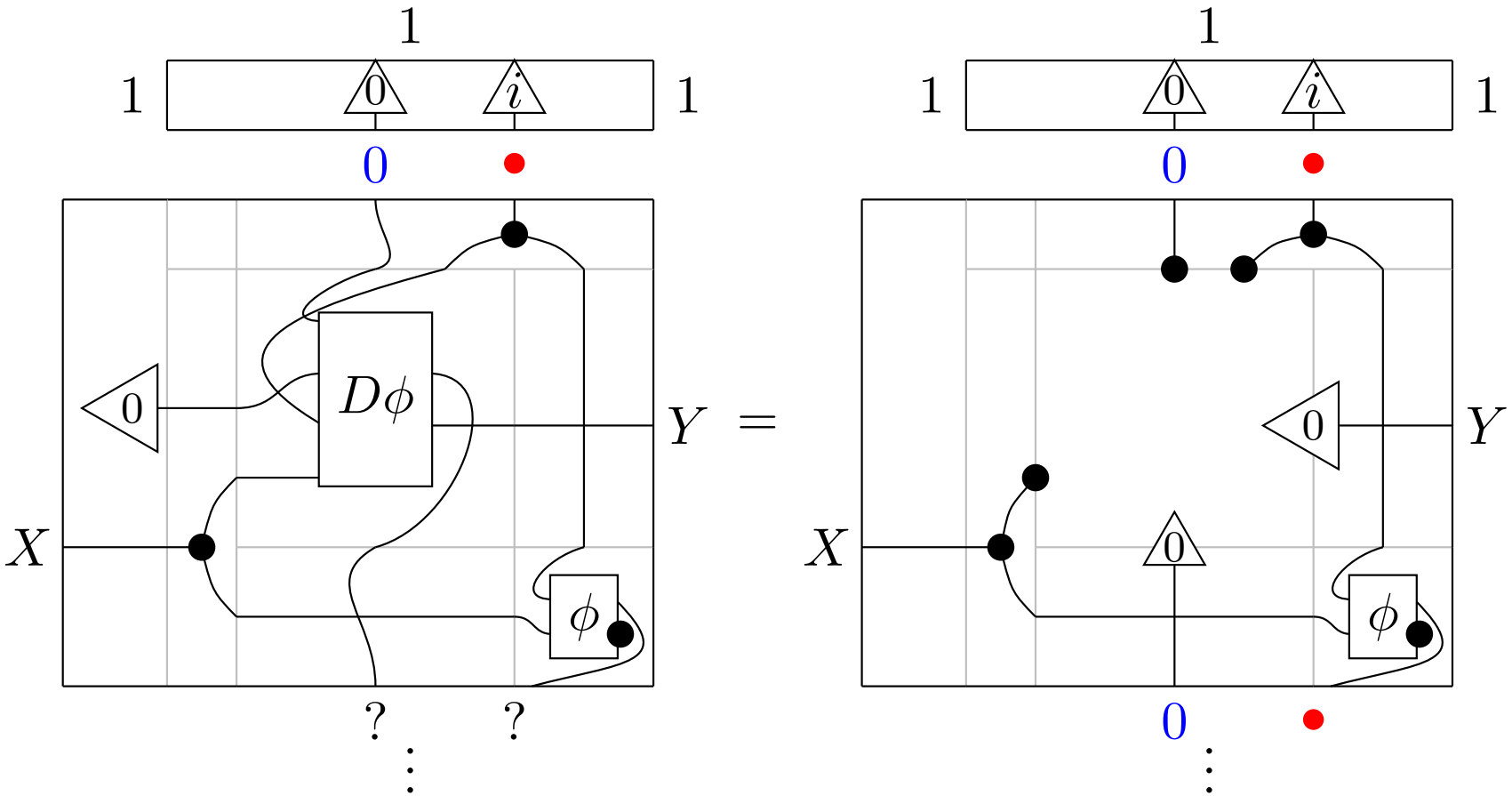




Key contribution: differential operator lifts

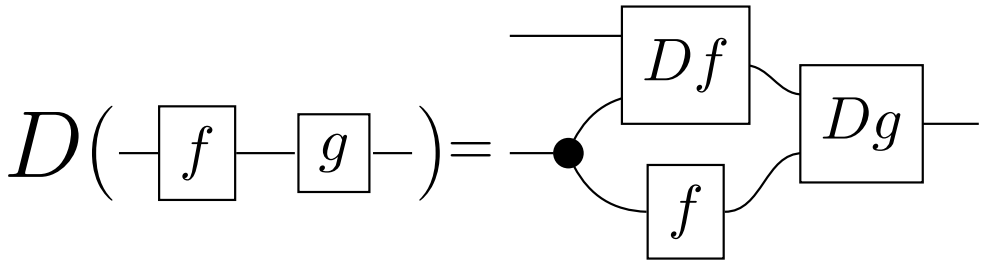
Proof idea. For CD2:

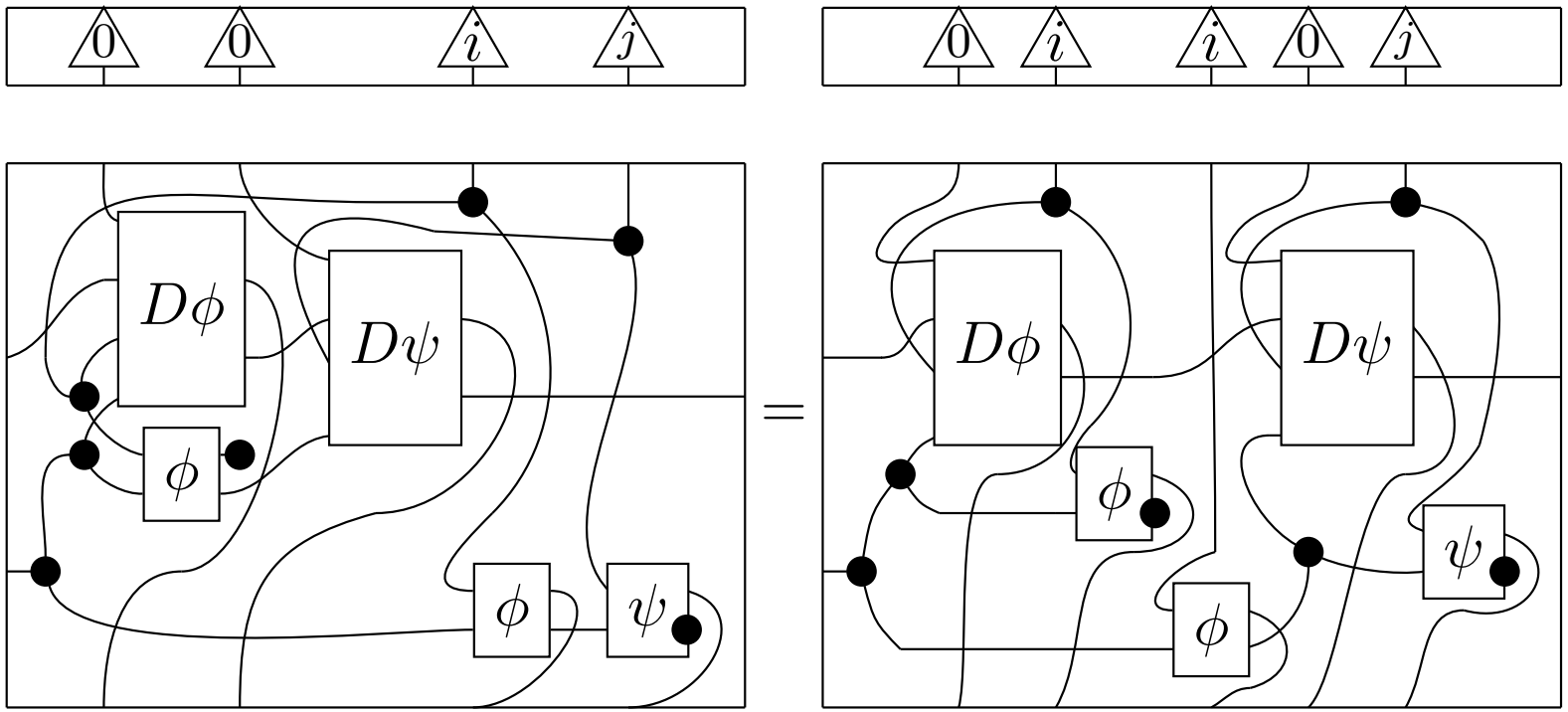




Key contribution: differential operator lifts

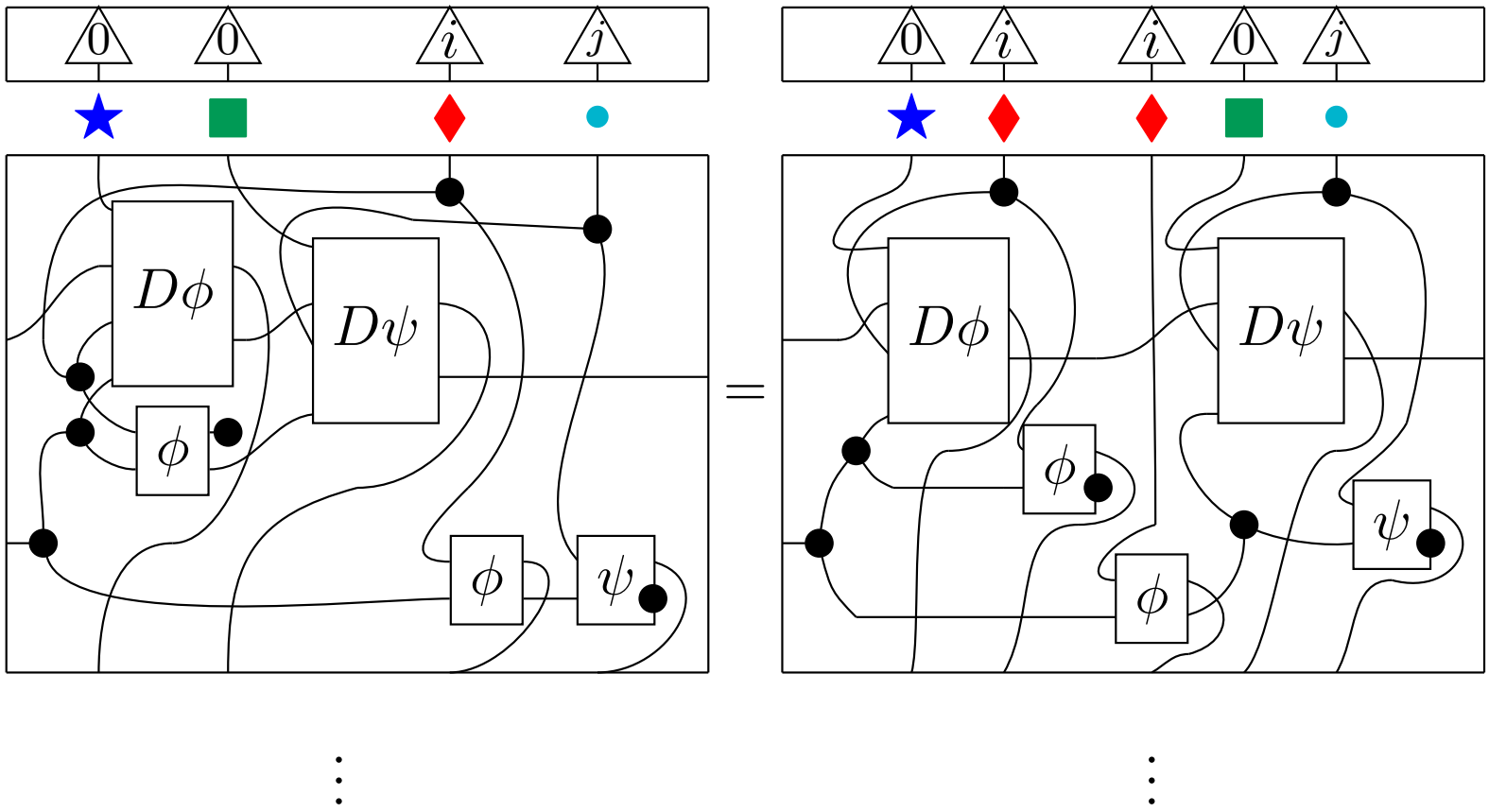
Proof idea. For CD4:





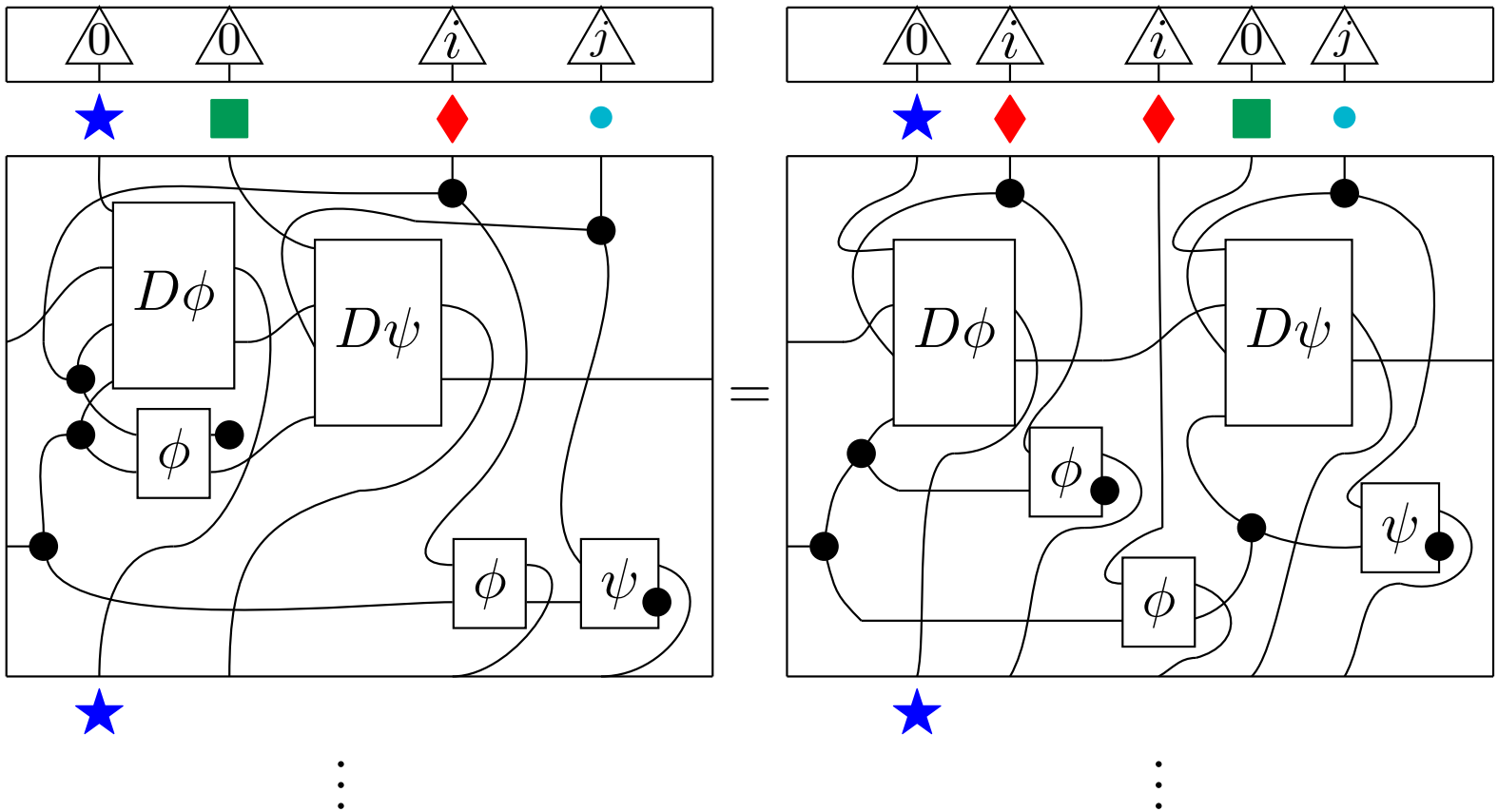
Key contribution: differential operator lifts

Proof idea. For CD4: $D(f \circ g) = Df \cdot f \circ Dg$

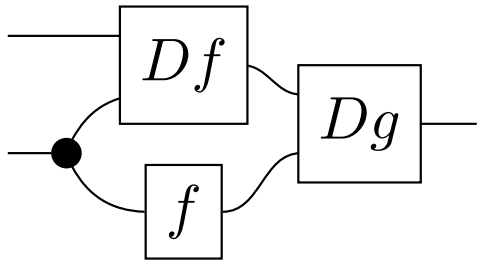


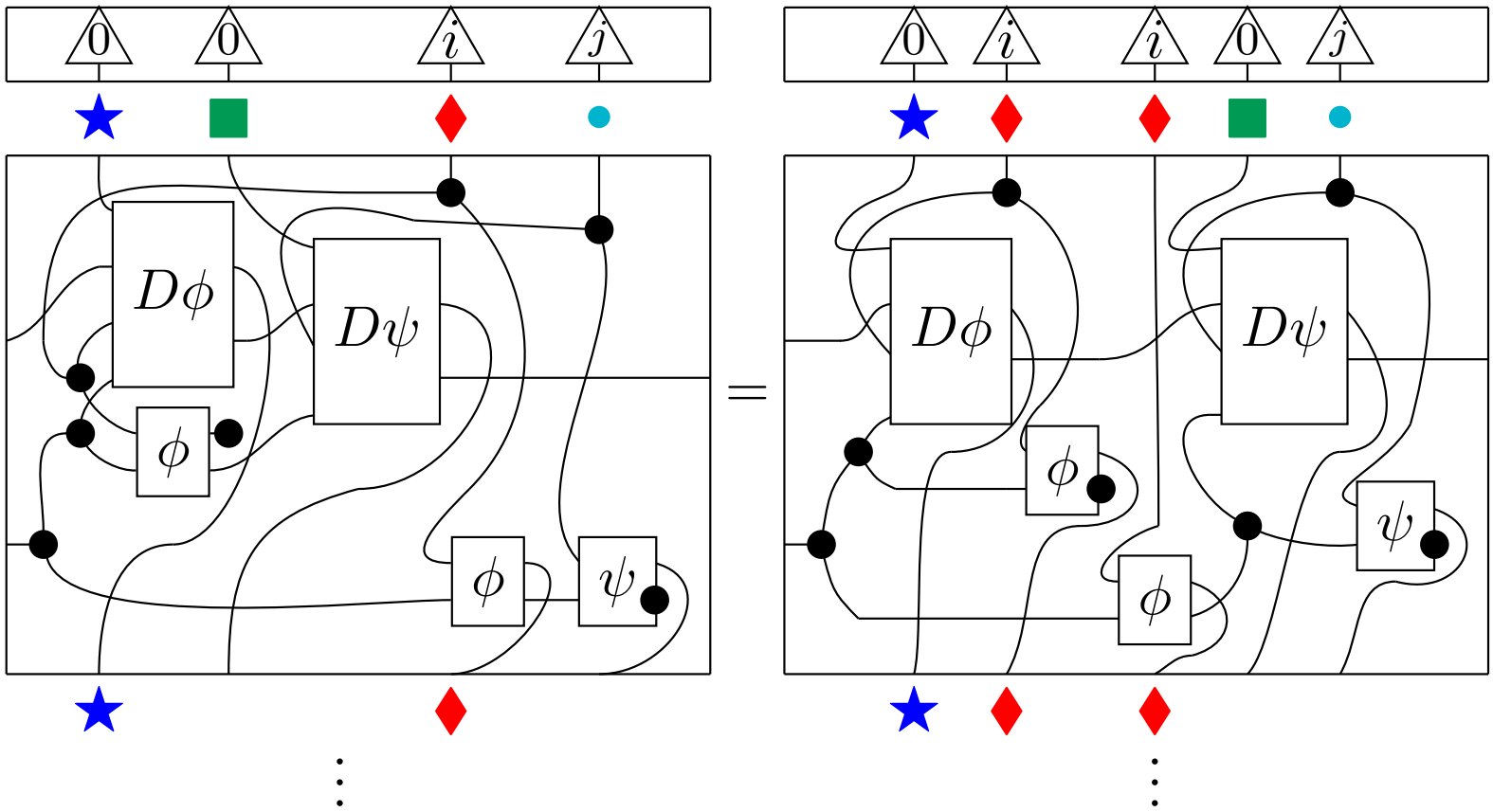
Key contribution: differential operator lifts

Proof idea. For CD4: $D(f\,g) = Df\,f\,Dg$

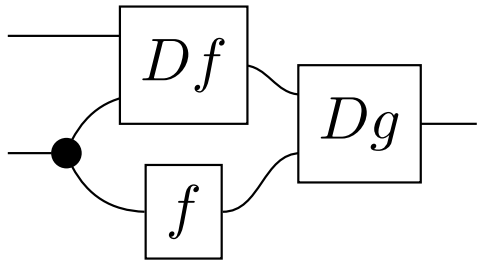


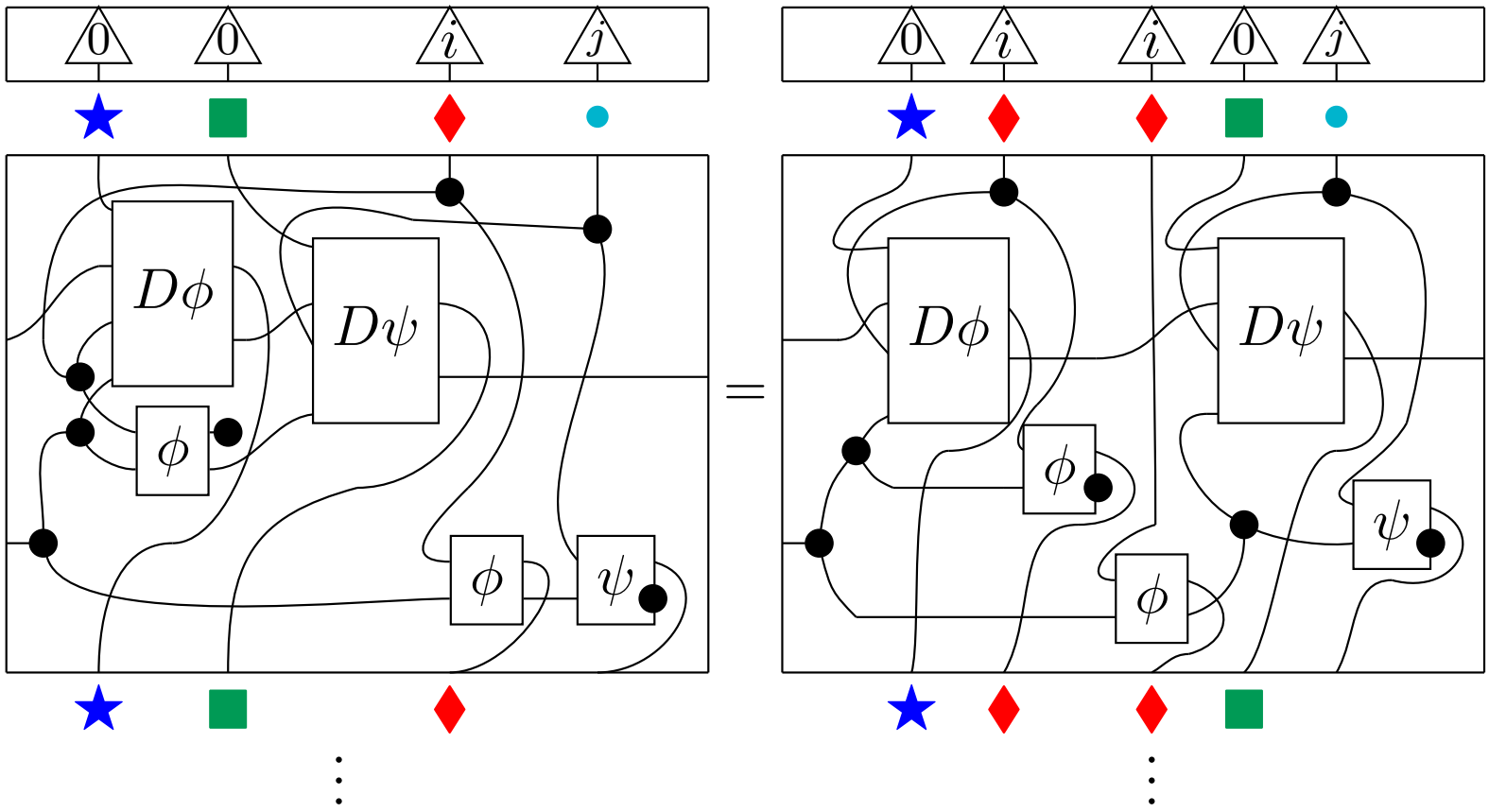
Key contribution: differential operator lifts

Proof idea. For CD4: $D\big(-\boxed{f}-\boxed{g}-\big)=$ 

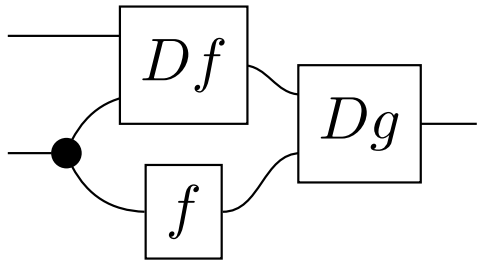


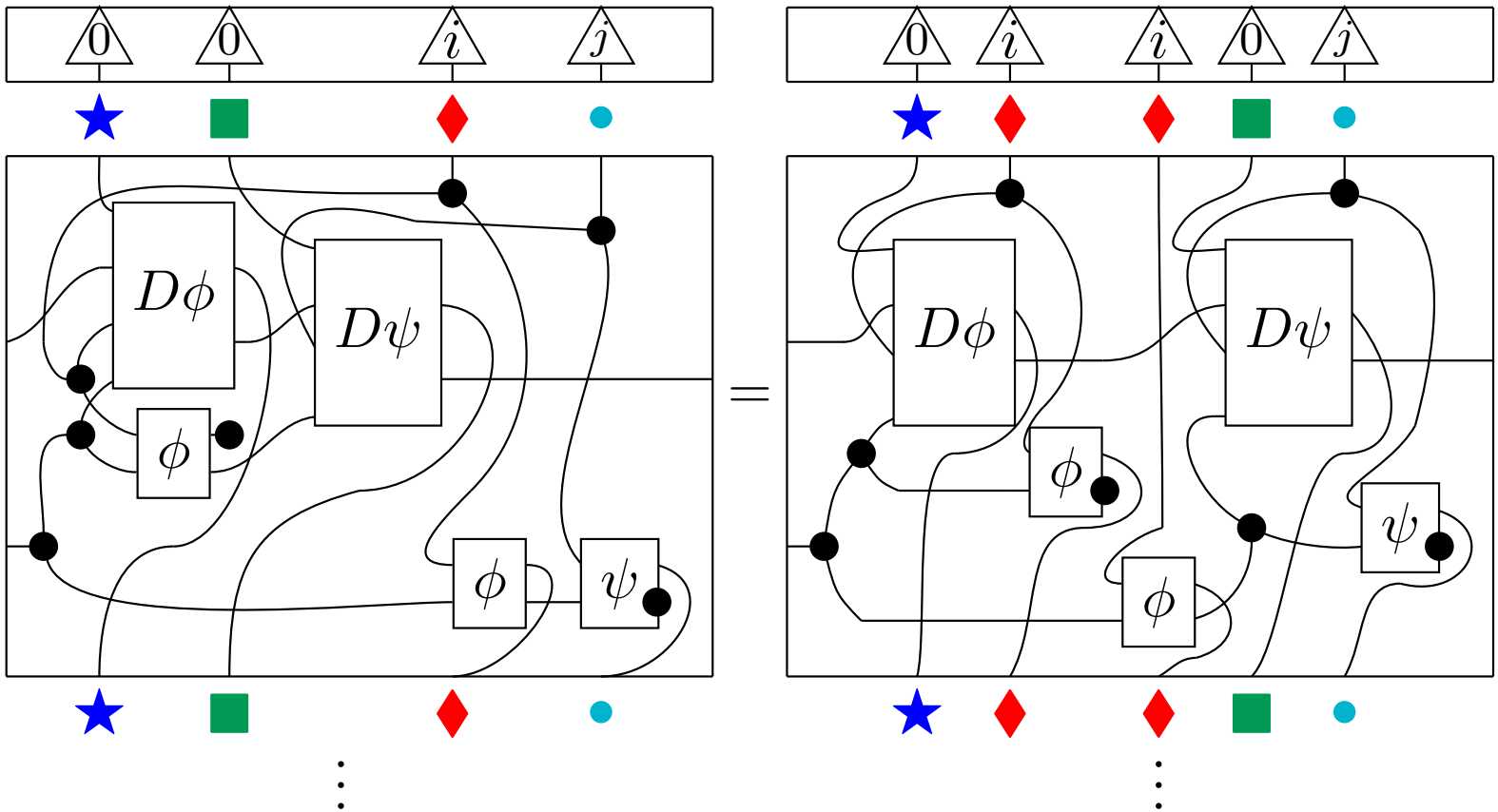
Key contribution: differential operator lifts

Proof idea. For CD4: $D\big(-\boxed{f}-\boxed{g}-\big)=$ 

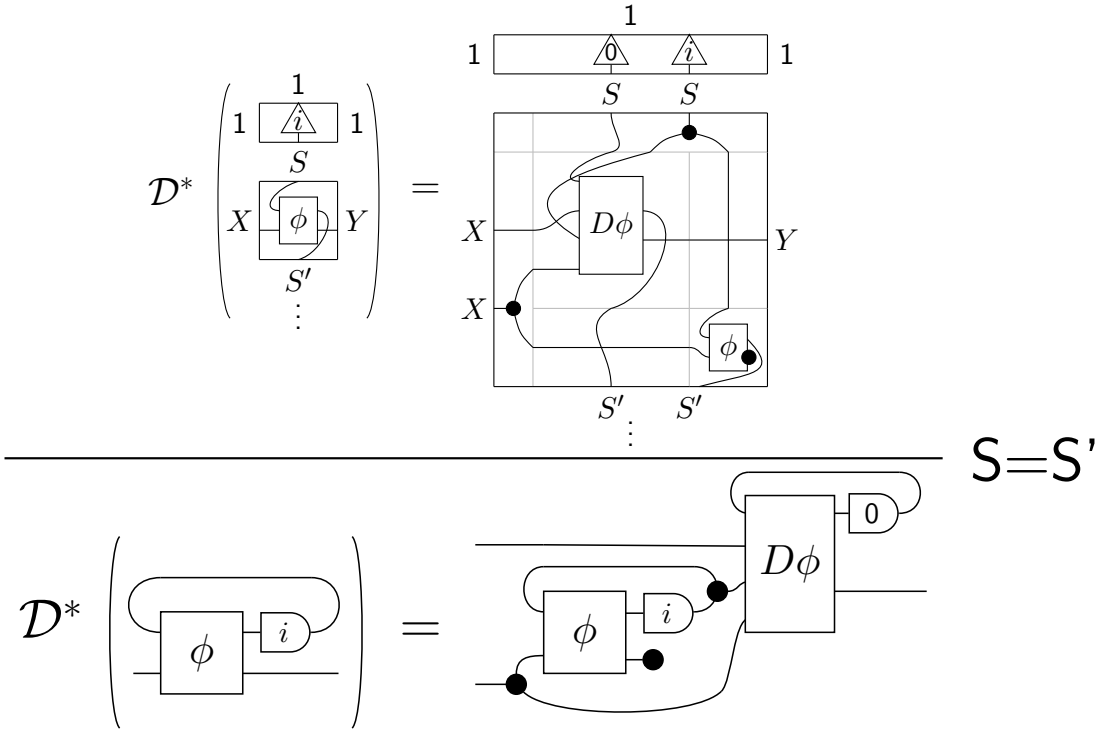


Key contribution: differential operator lifts

Proof idea. For CD4: $D\big(f\ g\big) =$ 



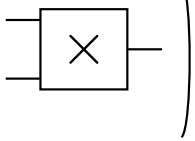
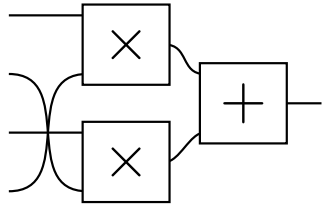
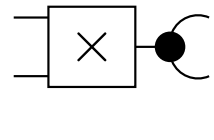
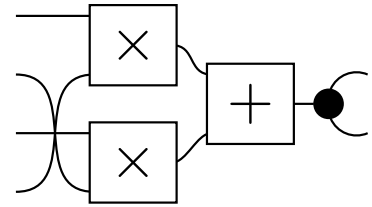
Differentiating circuits



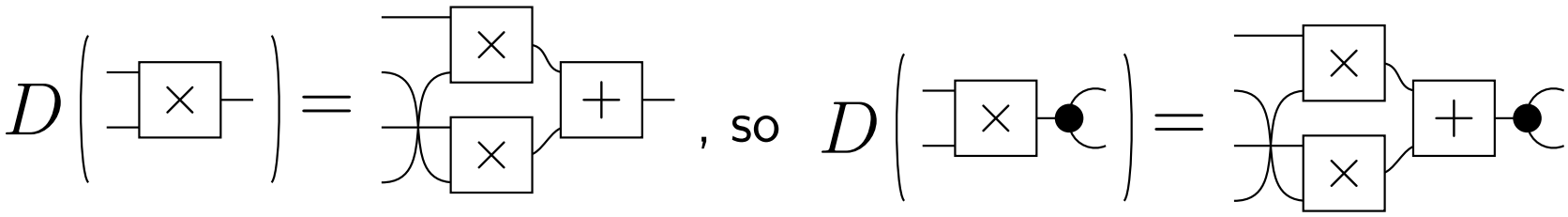
Theorem ($\mathcal{D}^*$ matches BPTT)

The unrolling of $\mathcal{D}^*((i, [\phi]))$ is the component-wise application of D to the unrolling of $(i, [\phi])$.

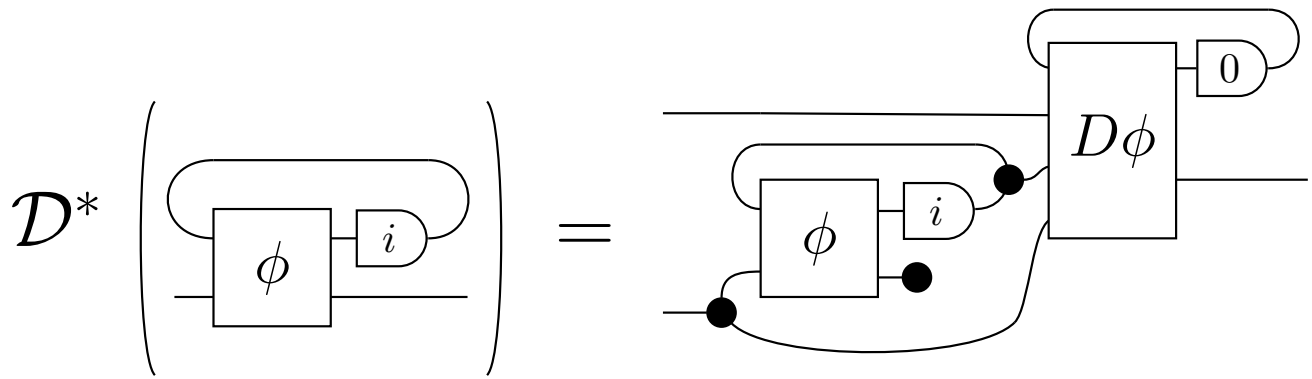
Where we're heading, update III

$D\left($  $\right) =$  , so $D\left($  $\right) =$ 

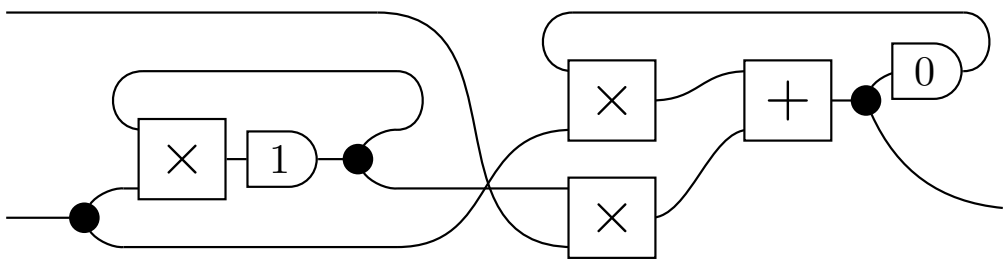
Where we're heading, update III



Then



becomes



Outline

Recap

We gave a mechanism for defining causal functions based on computation sequences. We organized these with a construction on Cartesian categories, $\mathrm{St}(-)$. These categories have a *delayed trace*.

We showed that if $\mathbb{C}$ is a Cartesian differential category, so is $\mathrm{St}(\mathbb{C})$. Taking $\mathbb{C} = \mathbf{Euc}_\infty$, we can find derivatives of recurrent neural networks. The differential operator on $\mathrm{St}(\mathbf{Euc}_\infty)$ matches the derivatives taken in backpropagation through time; the fact that it is a differential operator obtains many properties for BPTT.

Future questions (roughly mathematical to practical)

1. Categorical properties of $\mathrm{St}(-)$?
2. $\mathcal{D}^*$ and derivatives in sequence spaces?
3. Bisimulations and extensional equality?
4. Basic results for delayed trace categories?

Future questions (roughly mathematical to practical)

1. Categorical properties of $\mathrm{St}(-)$?
2. $\mathcal{D}^*$ and derivatives in sequence spaces?
3. Bisimulations and extensional equality?
4. Basic results for delayed trace categories?
5. Axiomatization of stateful function diagrams?
6. Explicit Jacobians using closed structure?
7. Probabilistic/nondeterministic causal functions?
8. Partial functions with differential restriction categories?

Future questions (roughly mathematical to practical)

1. Categorical properties of $\mathrm{St}(-)$?
2. $\mathcal{D}^*$ and derivatives in sequence spaces?
3. Bisimulations and extensional equality?
4. Basic results for delayed trace categories?
5. Axiomatization of stateful function diagrams?
6. Explicit Jacobians using closed structure?
7. Probabilistic/nondeterministic causal functions?
8. Partial functions with differential restriction categories?
9. Iterations I: RNN parameters in $\mathrm{St}(\mathrm{St}(\mathbb{C}))$?
10. Iterations II: RNN hyperparameters in $\mathrm{St}(\mathrm{St}(\mathrm{St}(\mathbb{C})))$?

Future questions (roughly mathematical to practical)

1. Categorical properties of $\mathrm{St}(-)$?
2. $\mathcal{D}^*$ and derivatives in sequence spaces?
3. Bisimulations and extensional equality?
4. Basic results for delayed trace categories?
5. Axiomatization of stateful function diagrams?
6. Explicit Jacobians using closed structure?
7. Probabilistic/nondeterministic causal functions?
8. Partial functions with differential restriction categories?
9. Iterations I: RNN parameters in $\mathrm{St}(\mathrm{St}(\mathbb{C}))$?
10. Iterations II: RNN hyperparameters in $\mathrm{St}(\mathrm{St}(\mathrm{St}(\mathbb{C})))$?
11. Measuring complexity with string diagrams?
12. What happens when $\phi \in \{LSTM, GRU, \ldots\}$?
13. Implementation/plugin to neural net library?

Thanks!