∂P.jl terminology and notation guide

Keno Fischer

FOREWORD

The field of automatic differentiation (AD) has a long history, both in the academic literature and in programming practice. However, despite this long history, or perhaps because of it, the terminology for various concepts and operations is neither standard nor consistent. This inconsistency imposes upon new implementers of AD tools a burden to clearly state the meaning of any terminology they might venture to employ. This document is an attempt to do such that for my present work on a new AD tool for the Julia language. It is intended 1) to give contributors and users a common vocabulary to discuss the manifold abstractions required to create a modern AD tools, 2) to explain why certain terminology may have been chosen and 3) to serve implementers as a guide to the proper functioning of the tool when lost deep in the web of details. By necessity, the terminology in this guide is borrowed from several different fields of study (most relevantly category theory and differential geometry). Nevertheless, this guide does not intend to be an introduction to these fields, nor does it assume that readers have had such an introduction. Instead, we try to give some examples and intuition for why the abstractions chosen are of interest to the implementation of automatic differentiation. Lastly, despite using terminology that might be amenable to formalization, this document is not an attempt to do so and as such certain liberties may be taken that would require careful treatment in a more formal treatise. With that out of the way, let's begin:

I. OPTICAL CONSTRUCTIONS

To begin our quest, we shall first put aside the topic of differentiation entirely and study a relatively recent construction from category theory known as an *optic*. The power of the optic construction lies in its ability to combine both a covariant and a contravariant transformation in one abstraction while retaining composability. Following Riley [1], we briefly consider the following explicit description of an optic (those not familiar with the language of category theory are encouraged to skip straight to the diagrams).

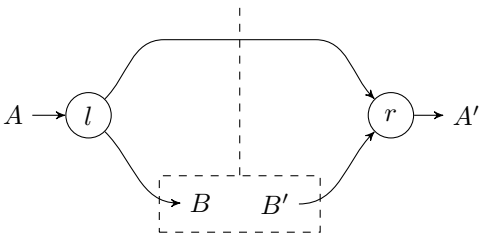
Definition 0.1: For some symmetric monoidal category $\mathcal{C}$, we define the category $\mathbf{Optic}_{\mathcal{C}}$. Whose objects are pairs (A, A') of objects in $\mathcal{C}$ and whose morphisms have representatives given by $\langle l|r\rangle : (A, A') \nrightarrow (B, B')$ are pairs (l, r) where $l : A \to M \otimes B$, $r : M \otimes B' \to A'$ (the choice of the object M depending on the representative - see Riley for details).

This definition makes manifest the combination of co- and contravariant data. For a representative $\langle l|r\rangle$, l varies covariantly while r varies contravariantly. We additionally have a "memory" or "residual" object M. This object is not uniquely determined and in fact we shall make good use of that fact in our actual use case later. For now, We briefly exhibit the composition rules for these morphisms. Suppose $\langle l_1|r_1\rangle \ : \ (A, A') \nrightarrow (B, B')$, $\langle l_2|r_2\rangle \ : \ (B, B') \nrightarrow (C, C')$. Then the sequenial composition $\langle l_{12}|r_{12}\rangle = \langle l_1|r_1\rangle \fatsemi \langle l_2|r_2\rangle$ may be written explicitly as

$$l_{12}(a) := (m_1 \otimes m_2, c) \text{ where } (m_1, b) = l_1(a), (m_2, c) = l_2(b)$$

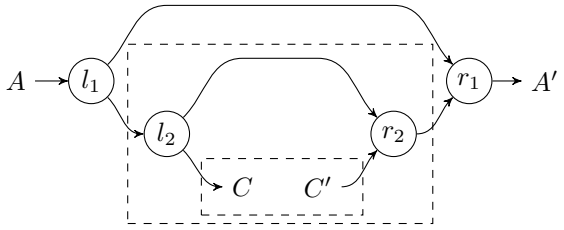
$$r_{12}(m_1 \otimes m_2, c') := r_1(m_1, r_2(m_2, c'))$$

While these definitions are perfectly explicit, it might be easier to understand what is going on by considering optics diagramatically (diagrams taken from Riley [1]). For $\langle l|r\rangle$ we draw:



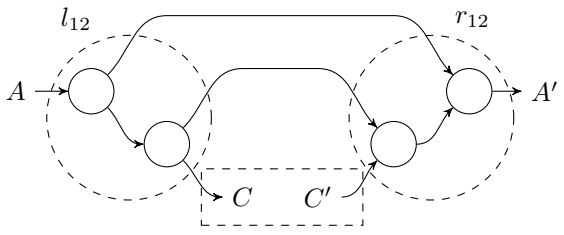
(1)

We compose these optics by inserting one optic in the hole of another, e.g. for the composition we have above, we might write $\langle l_1|r_1\rangle \fatsemi \langle l_2|r_2\rangle$



(2)

From the diagrammatic view, it is immediately clear that this composition is indeed another optic and what the composition rule for these optics should be:



(3)

The reader is encouraged to verify that this diagrammatic derivation of l_{12} and r_{12} matches the algebraic definition above.

A. Encoding Optics in Julia

One of the original applications of optic constructions is the manipulation of deeply nested data structures, so let's take that example and consider how we might represent it in Julia. Optics for this application are often called *lenses*, though since we are just using them an an example here we shall not concern ourselves with a precise definition. We begin by defining an abstract optic type as well as a concrete representation of morphisms as pairs $\langle l|r\rangle$:

Snippet 1

```
abstract type AbstractOptic; end

struct OpticRepr <: AbstractOptic
  l::Function
  r::Function
end
OpticRepr(o::OpticRepr) = o
OpticRepr(o::AbstractOptic) =
  OpticRepr(a->left(o, a),
    (m, b′)->right(o, m, b′))

left(o::OpticRepr, a) = o.l(a)
right(o::OpticRepr, m, b′) = o.r(m, b′)

function ⨟(o₁::AbstractOptic, o₂::AbstractOptic)
  OpticRepr(
    function (a)
      x = left(o₁, a)
      y = left(o₂, x[2])
      (x[1], y[1]), y[2]
    end,
    ((m₁, m₂), c′)->right(o₁, m₁, right(o₂, m₂, c′))
  )
end
```

Given these definitions, we can now define an optic constructor that allows us to modify an immutable value at a given location:

Snippet 2

```
SetIndex(idx) = OpticRepr(
    obj->(obj, getindex(obj, idx)),
    (obj, update)->setindex(obj, update, idx)
)
```

which we may use like so:

```
julia> tup = (1,(2,3))
julia> o = SetIndex(2) ⨟ SetIndex(1)
julia> m, c = left(o, tup)
julia> c
2
julia> right(o, m, c + 100)
(1, (102, 3))
```

However, as written, this definition as a big problem: It gives us the state 'm' explicitly. The type and value of 'm' depends on the representative of the optic, but nothing in our above definition requires the state 'm' to be matched to the optic representative 'o'. To see the representation problem explicitly, suppose we noticed that the state in this representation is somewhat suboptimal:

```
julia> m
((1, (2, 3)), (2, 3))
```

and instead defined a more optimized representation of the same optic:

Snippet 3

```
struct FastSetIndex <: AbstractOptic
    path::NTuple{N, Int} where N
end
FastSetIndex(i::Int) = FastSetIndex((i,))

path_getindex(obj, ::Tuple{}) = obj
path_getindex(obj, path::Tuple) =
    path_getindex(getindex(obj, first(path)), tail(path))
left(f::FastSetIndex, obj) =
    (obj, path_getindex(obj, f.path))

path_setindex(obj, update, (idx,)::Tuple{Int}) =
    setindex(obj, update, idx)
path_setindex(obj, update, path) =
    setindex(obj,
        path_setindex(getindex(obj, first(path)),
        update, tail(path)),
        first(path))
function right(f::FastSetIndex, obj, update)
    path_setindex(obj, update, f.path)
end

(a::FastSetIndex ⨟ b::FastSetIndex) =
    FastSetIndex((a.path..., b.path...))
```

Our example from above goes through just the same:

```
julia> tup = (1,(2,3))
julia> of = FastSetIndex(2) ⨟ FastSetIndex(1)
julia> m, c = left(of, tup)
julia> c
2
julia> right(of, m, c + 100)
(1, (102, 3))
```

but our state is much smaller:

```
julia> m
(1, (2, 3))
```

With these two different representatives for the same optic, the representative, dependence of 'm' becomes quite apparent:

```
julia> right(of, m, c + 100)
(1, (102, 3))

julia> right(o, m, c + 100)
ERROR: MethodError: no method matching
setindex(::Int64, ::Tuple{Int64,Int64}, ::Int64)
```

In this example, this might be quite silly, but in general it presents a problem depending on how the representative is chosen. E.g. an optic constructor might perform a randomized search for the most efficient representative, so two calls to the constructor will not necessarily yield the same representative. We would thus prefer to have an interface to our optics that either does not explicitly expose the state to the user, or if it does, tags the state with a particular choice or representative. Here are two (equivalent) interfaces to optics that have this property:

Snippet 4

```
# Block style optics interface
function (o::AbstractOptic)(f::Function, a)
    m, b = left(o, a)
    b′ = f(b)
    right(o, m, b′)
end

# Continuation style optics interface
function (o::AbstractOptic)(a)
```

```
10      m, b = left(o, a)
11      b, b′->right(o, m, b′)
12  end
```

With these definitions, our update operation from above becomes

```
julia> of(tup) do c
    c + 100
end
(1, (102, 3))

julia> c, back = of(tup);

julia> back(c+100)
(1, (102, 3))
```

The observant reader may at this point remark that written this way, we are simply describing a standard design pattern, found frequently in julia libraries (indeed so frequently that the *do* syntax was partly introduced to support it). For example, the julia standard library includes the *cd* function, which modulo error handling function looks like this:

```
function cd(f::Function, dir::AbstractString)
    old = pwd()
    cd(dir)
    f()
    cd(old)
end
```

While it's not a precise match for the optics encoding as we have defined it (and of course is side-effectful which we have not discussed at all).

```
OpticRepr(a->(pwd(), (cd(a[1]); tail(a))),
          (m, b)->cd(m)))
```

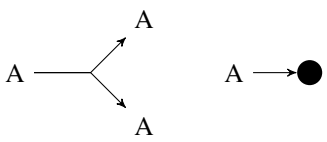
Have we thus wasted the better part of two pages exploring and giving a name to a notion that is so natural as to be a standard design pattern? No! The key takeaway from this section should be twofold optics: 1) Optics have a nice composition property that allows you to take two optics and combine them into another optic and 2) Optical constructions are quite natural and you are probably already familiar with them.

II. SSA as a category / Optics over SSA

In Julia's optimizer, functions are represented by their single static assignment (SSA) form. We shall venture to sketch a categorical description of SSA form and then investigate the structure of Optic over such a category.

A. Duplication / Deletion of Values

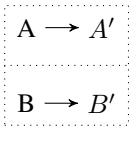
In our examples, so far every values was used once and exactly once. In SSA form, we may have an arbitrary number of uses. To support this, we extend our category with the diagonal $\Delta_{SSA} : A \to A \otimes A$) and deletion ($\dashv$: $A \to I$) morphisms. Graphically, we have:



(4)

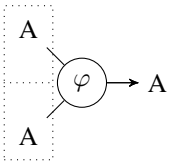
B. Forward control flow

To account for control flow, we add a coproduct to our category, which we shall denote by $\oplus$. Note that this is different from the $\otimes$ operation, we have so far considered, and we are thus creating a **bimonoidal** category, with two separate monoidal structures. Graphically, we will indicate $\oplus$ by boxes around the operations in questions, so for example, for the morphism $(f \oplus g) \otimes h : (A \oplus B) \otimes C \to (A' \oplus B') \otimes C'$, we will draw:



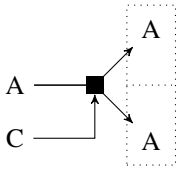
(5)

Now, we have an operation $\varphi : A \oplus A \to A$, that joins control flow edges:



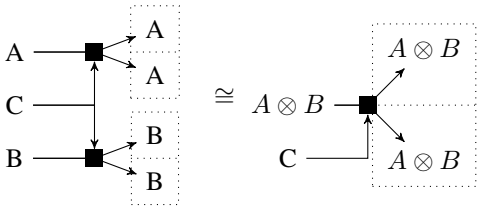
(6)

and we will introduce an additional "conditonal branch" operation $br : A \otimes C \to A \oplus A$ (where we say C is the condition):



(7)

We further impose the following *co-predication relation*:



(8)

As a notational convenience, we write $\oplus_c$ for the coproduct introduced under some abstract condition c. In that case, we may write the co-predication condition as

$$\leftrightarrow_{\oplus} := (A \oplus_c B) \otimes (C \oplus_c D) \cong (A \otimes C) \oplus_c (B \otimes D)$$

To avoid running into technical difficulties, we further allow commuting arbitrary morphisms through the branch operations, such that we can be sure every $A \oplus B$ is always potentially the direct result of a branch operation, such that we may validly introduce abstract condition tags on any such $\oplus$ operation [1].

[1] I realize this is a bit handwavy. While I believe this works fine, I would be interested in improvements to make this construction more formal and potentially avoid the reliance on commuting morphisms through the branch operation so as ot facilitate the future addition of side effects.

C. Lifting to the Optic category

1) Product Structure: Having defined the structure for our bimonoidal category $SSA_{\mathcal{C}}$, what can we say about the structure of $\text{Optic}_{SSA_{\mathcal{C}}}$? To be explicit, we're still performing the optic construction with respect to the $\otimes$ structure of our category. Let us first look at the product itself. Do we have a diagonal map in Optic_{SSA}? Perhaps surprisingly, the answer is no! To see this, let's consider what we must construct:

We wish to find a map $\text{Optic}(\Delta_{SSA}) : (A, A') \to (A, A') \otimes (A, A')$. A representative for such a morphism would have the form $\langle l, r\rangle$: $l : A \to M \otimes A \otimes A$, $r : M \otimes A' \otimes A' \to A'$. Finding l is easy, we can just use Δ_{SSA} (letting $M = I$) from the underlying category. However, there is no good way to construct r. A priori, there is no reason to expect any morphism of the shape $A' \otimes A' \to A'$ to exist, and even if such morphisms do exists, there is no reason to expect them to have the required uniqueness properties. It is important to note that this does not prevent us from lifting Δ_{SSA} to Optic in our applications of interest, it simply means that $\otimes$ is not a categorical product in $\text{Optic}_{SSA_{\mathcal{C}}}$, so we must make make a choice of summation morphism in our lifting functor.

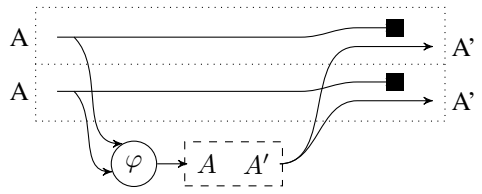
2) Coproduct Structure: Given our utter disappointment with the product structure, can we have any hope to lift the co-product structure. Yes, we do! First we construct the co-product itself. For two optics $\langle l_1|r_1\rangle : (A, A') \to (B, B')$ with residual M_1 and $\langle l_2|r_2\rangle : (C, C') \to (D, D')$ with residual M_2, we construct a new optic $\langle l_{12}|r_{12}\rangle$ where

$$\begin{aligned} l_{12} &= (l_1 \oplus l_2)\fatsemi \leftrightarrow_{\oplus} \\ r_{12} &= \leftrightarrow_{\oplus}^{-1} \fatsemi (r_1 \oplus r_2) \end{aligned} \tag{9}$$

Similarly, we can confirm that this is indeed a coproduct by exhibting the φ morphism. The trick here is to use the residual to carry the branch history information. Perhaps the simplest representative of $\text{Optic}(\varphi) : (A \oplus_a A, A' \oplus_a A') \to (A, A')$ would be $\langle l|r\rangle$ where

$$\begin{aligned} l &= \Delta_{SSA} \fatsemi (\text{id}_{A \oplus_a A} \otimes \varphi) \\ r &= d \fatsemi (\pi_2 \oplus \pi_2) \end{aligned} \tag{10}$$

where d is the distributive map. While this may be a little abstract, hopefully it becomes clear as a picture:



(11)

Note that another representative of the same optic could simply record the active component (e.g. by using the map $(x \to 1) \oplus (y \to 2)$ and reintroducing a branch on the RHS).

However, note that (at least the way we chose to set it up) we do not have a unique lifting of the branch operator itself.

III. Optic Functors

One useful way to construct optics is to do so automatically, from simpler descriptions. More formally, we are interested in functors $F : \mathcal{C} \to \textbf{Optic}_{\mathcal{D}}$. The requirement for F to be a functor means that composition in $\mathcal{C}$ matches composition in $\textbf{Optic}_{\mathcal{D}}$, i.e. $F(f)\fatsemi F(g) = F(f \fatsemi g)$. We shall call such functors (whose codomain is an optic category) *optic functors*.

We have already seen an example of this kind of operation! In our definition of *FastSetIndex* above, we represented the optic, as a path of indices. If we wanted to, we could consider this path of indices description a category (e.g. the category in which objects are Julia datatypes and there is morphisms from a datatype to each of its field types). In this case, the optic functor is essentially the definitions of the *left* and *right* generic, functions. Of course, we could have also left off the subtype and instead defined a constructor of *OpticRepr* with the *left* and *right* definitions inline (such a constructor does exist in our above definition, by virtue of the subtype). In that case, we might want to check the functor law like so:

```
julia> o₁ = OpticRepr(FastSetIndex(2) ⨟
                      FastSetIndex(1))
julia> o₂ = OpticRepr(FastSetIndex(2)) ⨟
            OpticRepr(FastSetIndex(1))
julia> o₁(c->c+100, tup)
(1, (102, 3))

julia> o₂(c->c+100, tup)
(1, (102, 3))
```

However, it is import to emphasize that while optic functors are required to preserve composition, when implemented as above, they are **not** required to produce the same representative. Indeed, in our example:

```
julia> o₁.l(tup)[1]
(1, (2, 3))

julia> o₂.l(tup)[1]
((1, (2, 3)), (2, 3))
```

the two representatives o_1 and o_2 are not identical, even though they represent the same optic.

A. Optic functors on programs

We now leave behind our trusty *setindex* example and consider optic functors from programs. To keep the discussion somewhat concrete, consider the following example: Alice, Bob, and Clara are department heads at Consolidated Widgets Incorporated. One day the anti-trust regulator decides that Consolidated Widgets Inc is a little too consolidated and trisects the company into AliceCo, BobCo and ClaraCo with the three former department heads promoted to the CEOs of their respective companies. They now have a problem: While they know how to manufacture widgets, they now also need to keep track of how much each individual part costs and assign profits back to the appropriate company. Can optic functors help? Yes!

Suppose the program for widget assembly is simply

```
widget(sweat, love) = C(A(sweat), B(love))
```

i.e. Alice and Bob both manufacture subassemblies from raw inputes (sweat and love) and Clara handles the final

assembly). We wish to construct an optic functor € that takes in the widget construction function and spits out a function that both constructs the widgets and accumulates prices on the left side of the optic, and then distributes the profits on the right side of the optic after the widgets have been sold. Now, we have four function of interest here $widget$, A, B and C. For each of A, B and C, we need to do some manual work. Alice, Bob, and Clara need to do some manual work to figure out what their costs are. We say that A, B and C are *primitive* with respect to this functor. Alright, let's suppose we have our prices (for simplicity 10€ for Alice, 20€ for Bob and 30€ for Clara). We will also give each of them a flat 30% profit margin. We will use the continuation encoding of the optic to specify the value of € on each of our primitives:

Snippet 4

```
€A(price, sweat) = let cost=10;
  (price + cost, A(sweat)),
    profit->(profit - cost * 1.3)
end
€B(price, love) = let cost=20;
  (price + cost, B(love)),
    profit->(profit - cost * 1.3)
end
```

C is a little more tricky. For C, we have two arguments, so on right side of the optic, we need to produce two outputs. There are several ways to model this. We could consider only allowing function with a single input and a single output, together with primitive operations for packing (into a tuple) and unpacking, or we could simply require the output to always be a tuple whose length matches the original number of arguments. Here we pick the latter encoding and write:

Snippet 5

```
€C(aprice, bprice, a, b) = let cost=30;
  (aprice + bprice + cost, C(a, b)),
    profit->(take(profit, aprice),
             take(profit, bprice))
end
```

Ok, phew that was a bit of work to figure out all the primitives. However, now that this is done, we do get some reward for our efforts. We get the definition of €$widget$ for free by applying the functor law. Recall that we essentially defined $widget = (A \otimes B) \fatsemi C$. Thus, applying the functor law, we immediately get that €$widget = (€A \otimes €B) \fatsemi €C$

By tracing through definitions, we can write out an explicit representative in the continuation encoding:

Snippet 6

```
function €widget(sweat_cost, love_cost, sweat, love)
  aprice, a, a* = €A(sweat_cost, sweat)
  bprice, b, b* = €B(love_cost, love)
  price, c, c* = €C(aprice, bprice, a, b)
  price, c, profit->begin
    (a†, b†) = c*(profit)
    love† = b*(b†)
    sweat† = a*(a†)
    (sweat†, love†)
  end
end
```

Now, this might seem a bit complicated (and it is), but that is precisely the point. Writing this out explicitly is quite tedious, but there are no choices to be made in how it is written. **Once we have defined the optic functor on the requisite primitives, we can make use of the composition law to generate a representative of the functor applied to any function that is a composition for our primitives.**

B. Some notation

As we saw in the previous section, the explicit representatives generated by optic functors can be quite complicated. To avoid getting lost, we should make some notational conventions. To begin with, we shall consider input programs to be represented by their SSA form representation. We shall assume that the reader is familiar with SSA form IR (if not, see the Julia devdocs on SSA form IR). Now, let € be an optic functor. In the context of applying € to some function f, we shall say that f is the *primal function* and similarly we shall call the value computed by f the *primal value*. From now on, we will generally be working with the continuation encoding of an optic. By abuse of notation, we will write €f for the function that is the canonical representative of the optic in continuation encoding (canonical here meaning obtained by straightforward application of the functor laws). We shall call this function the *optic function*. We shall call the value returned from the optic function the *focal bundle*. Since we sometimes want to look at it in isolation, we will call the continuation generated by the optic function, the *pullback function*, written $f^\star$ for some primal function f. Since $f^\star$ depends on the representative, unless otherwise specified, $f^\star$ shall refer to the pullback function generated by the canonical optic function.

When looking at SSA form, IR, for some SSA value $\%a = A(...)$ in the primal function, we will write $(\%a, \%a^\star) =$ €$A(...)$ in the optic function. The intended implication here is that $\%a$ in the optic function is the same value as $\%a$ in the primal function. This will generally be the case, as long as the optic primitives obey this constraint. However, we have not required this invariant on primitives in our definition of the optic functor and there are (rare) cases in which it is useful for these to be different. Where the distinction matters, the context should be explicitly specified. Lastly, we look at the pullback function. By the optic composition law, pullbacks behave contravariantly to the primal function, so for a call $c = C(A, B)$ in the primal function, we will have a call $(\%a^\dagger, \%b^\dagger) = \underline{\%c^\star}(\%c^\dagger)$. Where in general an underline indicates that a value was captured from the optic function.

C. Optic functors from SSA IR

When we discussed SSA form in section II, we identified several structural features of the IR that did not lift uniquely to the optic category. In particular, this means that we have some freedom in choosing how these should lift for our optic functor. In particular, the structural aspects of SSA IR that did not have a unique lift were:

- Copy
- Delete
- Conditional Branch

As an exercise, let us go through these and choose an implementation.

1) Copy: In our little widget manufacturing example from the previous section, the objects being propagated in the primal functions were in general intermediate products that could only be used. But what happens if we introduce trivially copyable goods into the equation? E.g. suppose Alice and Bob both gain a trivially copyable input (perhaps music they like to listen to, or maybe they both require a license for JuliaPro), and that Dave produces such an input. For simplicity, we will also drop Alice and Bob's other inputs. We have a new program:

Snippet 7

```
function clonewidget()
    d = D()
    a = A(d)
    b = B(d)
    c = C(a, b)
end
```

Let us look at some sensible policy choices[2] we might make. For example, we could decide that David gets to decide what to charge for each copy of 'd':

Snippet 8

```
function €clone(dprice, d)
    (dprice, dprice), (d, d),
    (profit₁, profit₂)->(profit₁ + profit₂)
end
```

or maybe everybody pays a fixed fee, independent of David's costs:

Snippet 9

```
function €clone(dprice, d)
    (1, 1), (d, d),
    (profit₁, profit₂)->(profit₁ + profit₂)
end
```

or maybe everybody pirates his music and David gets payed nothing:

Snippet 10

```
function €clone(dprice, d)
    (0, 0), (d, d),
    (profit₁, profit₂)->0
end
```

or maybe the participants of our economy are playing hot potato, and always exactly one of the participants needs to pay:

Snippet 11

```
function €clone(dprice, d)
    rand(Bool) ? (dprice, 0) : (0, dprice), (d, d),
    (profit₁, profit₂)->(profit₁ + profit₂)
end
```

However, note that while this is a valid definition under our definition of an optic functor, applying € now leads to accumulation order dependence (the same happens in the variant where cloning is done once per value). As a result, € would no longer preserve standard SSA invariants. This is legal according to our definition, but it can be convenient to be able to arbitrarily permute SSA transforms and optic functors. Thus, we would generally only ever choose one of the first two definitions.

[2]Welcome to Copyright 101 - Today's topic: Category theory.

2) Delete: Like copy, we have some freedom what do here. For our example, it would probably make sense to dynamically disallow deleting objects that have a non-zero price, so we might write:

Snippet 12

```
function €delete(dprice, d)
    dprice != 0 && error("Who's gonna pay for that?")
    ()->(0,)
end
```

3) Branch: Again we have several potential choices. The primary reason that the branch operation was non-canonical was because of the implicit delete of the branch condition. However, it is possible for us to implement more complicated behaviors. For example suppose we wanted to proportionately assign to each input. We might write something like (encoding for the branch functor is slightly tricky, this is one example):

Snippet 13

```
function €branch(aprices, cprice, as, c)
    # Smear the cost of `c' proportionately
    total_a = sum(aprices)
    aprices = map(p->p*(1 + cprice/total_a), aprices)
    aprices, (c, as), profit->begin
        (profit - cprice), cprice
    end
end
```

However, as with the prior discussion, this would not match standard ssa behavior (i.e. programs equivalent under SSA would not longer be equivalent when lifted). As before, this is legal, but may not be desirable.

IV. Tangent spaces

We will briefly define the concepts relevant to us, but otherwise defer to any introductory text on differential geometry [3]. Let M be a smooth manifold, $C^\infty(M)$ the space of smooth functions on M. A tangent vector ν to M at a is an $\mathbb{R}$-linear derivation from $C^\infty(M)$ into $\mathbb{R}$, i.e. a function from $C^\infty(M) \to \mathbb{R}$ satisfying:

$$\nu(f \cdot g)(x) = g(a) \cdot \nu(f) + f(a) \cdot \nu(g) \tag{12}$$

The collection of all tangent vectors to M at a form a vector space, called the tangent space to M at a, T_aM. Additionally, if we glue all these vector spaces together, we obtain the tangent bundle TM.

We can dualize this vector space and obtain the cotangent bundle $T^\star M$. Sections of the cotangent bundle are also called differential 1-forms.

If we have a function between manifolds, say $f : M \to N$, we obtain a pushforward of tangent spaces: $f_\star : T_M \to T_N$ given by:

$$(f_\star \nu)g = \nu(g \circ f) \tag{13}$$

(for $g : C^\infty(M)$), as well as a pullback of cotangent spaces $f^\star : T^\star N \to T^\star M$ given by

$$(f^\star dx)\nu = dx(f_\star \nu) \tag{14}$$

[3]e.g. https://faculty.math.illinois.edu/ lerman/518/f11/8-19-11.pdf

These notions are generalizations of some of the standard notions of derivatives that may be familiar from multivariable calculus. Suppose $f : \mathbb{R}^n \to \mathbb{R}^m$, Let π_i be the component projection function. We obtain a canonical basis, of the tangent space $T\mathbb{R}^n$ given by the unique functions such that

$$\nu_i(\pi_j) = i == j \tag{15}$$

And similarly for the cotangent space.

Now the Jacobian Jf defined by

$$(Jf)_{ij} = \frac{\partial f_i}{\partial x_j} \tag{16}$$

is just matrix obtained by pushing forward the standard basis fo $T\mathbb{R}^n$ using $f_\star$. Equivalently, it is the transpose of the pullback of the standard basis of $T^\star\mathbb{R}^m$.

As a special case suppose $m = 1$. Then we have a function $\nabla f : \mathbb{R}^n \to \mathbb{R}$, the gradient of f at x. Using our language, this is simply the pullback of the differential one form dx_1 from $T^*\mathbb{R}$.

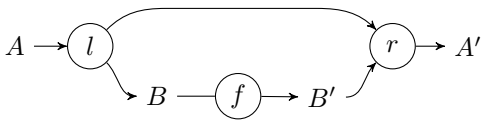
V. Advanced Optics considerations

A. Action of an Optic on the underlying category

The category $\mathbf{Optic}_{\mathcal{C}}$ acts on the underlying category $\mathcal{C}$ in the following sense: We have a contravariant functor $p : \mathbf{Optic}_{\mathcal{C}} \to \text{Set}$ that assigns to every object (A, A') the set $Hom(A, A\prime)$ and to morphisms $\langle l|r\rangle$ the morphism $p\langle l|r\rangle : Hom(B, B\prime) \to Hom(A, A\prime)$ given by:

$$p\langle l|r\rangle f = l \fatsemi (id_M \otimes f) \fatsemi r$$

or graphically by assigning to a morphism $f \in \text{Hom}(B, B')$ the morphism obtained by plugging f into the "hole" of the optic:



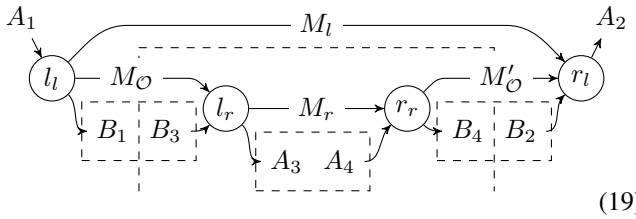
(17)

B. Higher order optics

As hinted at above and proven in detail in Riley [1], **Optic** is an endofunctor on the category of symmetric monoidal categories, i.e. for a symmetric monoidal category $\mathcal{C}$, $\mathbf{Optic}_{\mathcal{C}}$ is once again a symmetric monoidal category. In particular, this means we may repeat the application of **Optic** to obtain higher order optics. Let us explicitly characterize second order optics such that we may discuss some features that will come in later. Objects in $\mathbf{Optic}_{\mathbf{Optic}_{\mathcal{C}}}$ are pairs of objects in $\mathbf{Optic}_{\mathcal{C}}$, i.e. in general quadruplets of objects in the original category. We will write $\begin{pmatrix} (A_1 & A_2) \\ (A_3 & A_4) \end{pmatrix}$ for the higher order optic that pairs the objects $\begin{pmatrix} A_1 \\ A_2 \end{pmatrix} \begin{pmatrix} A_3 \\ A_4 \end{pmatrix}$ from the original optic category. Then a representative of the optic morphisms $\mathbf{Optic}\left(\begin{pmatrix} (A_1 & A_2) \\ (A_3 & A_4) \end{pmatrix}, \begin{pmatrix} (B_1 & B_2) \\ (B_3 & B_4) \end{pmatrix}\right)$, is a pair of optics $\langle \mathcal{O}_l | \mathcal{O}_r \rangle$ where

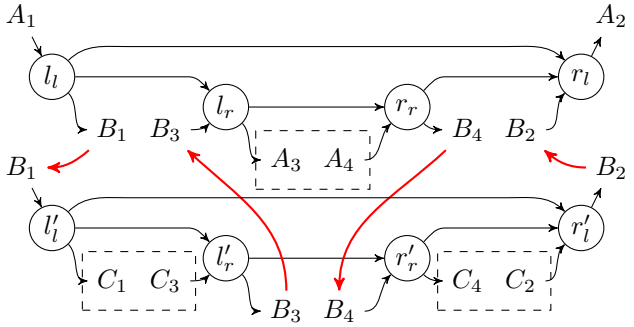
$$\begin{aligned} \mathcal{O}_l &\in \mathbf{Optic}\left(\begin{pmatrix} A_1 \\ A_2 \end{pmatrix}, \begin{pmatrix} M_{\mathcal{O}} \\ M'_{\mathcal{O}} \end{pmatrix} \otimes \begin{pmatrix} B_1 \\ B_2 \end{pmatrix}\right) \\ \mathcal{O}_r &\in \mathbf{Optic}\left(\begin{pmatrix} M_{\mathcal{O}} \\ M'_{\mathcal{O}} \end{pmatrix} \otimes \begin{pmatrix} B_3 \\ B_4 \end{pmatrix}, \begin{pmatrix} A_4 \\ A_4 \end{pmatrix}\right) \end{aligned} \tag{18}$$

. We take representatives $\langle l_l | r_l \rangle$ of $\mathcal{O}_l$ and $\langle l_r | r_r \rangle$ of $\mathcal{O}_r$ with residuals M_l and M_r respectively. Writing the morphisms out algebraically is a bit confusing, so let's look at the graphical representation:



(19)

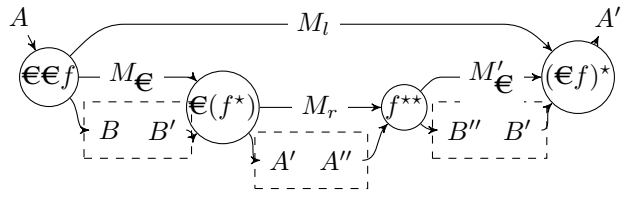
Here the dashed line plays the same role it did in the original diagram of the optic: To compose optics, we "slice" along the dashed line and insert the next higher order optic in the middle. To illustrate the composition, we will first look at two diagrams stacked on top of each other using the same structure as in the previous diagram, to make the flow of operations obvious.



(20)

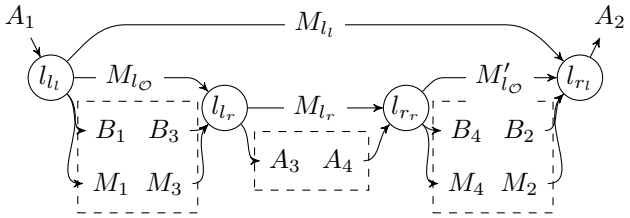
We can relayout this to make the four segments more obvious. This is done in figure 1. Now, if $\mathbb{C} : \mathcal{C} \to \mathbf{Optic}_{\mathcal{C}}$ is an optic functor, then since it is monoidal functor, it immediately lifts to a functor: $\mathbb{C} : \mathbf{Optic}_{\mathcal{C}} \to \mathbf{Optic}_{\mathbf{Optic}_{\mathcal{C}}}$. Note that optics constructed in this way will generally be more constrained than the full optics construction. For example, the 2-optic we constructed above has objects that are in general 4-tuples $((A_1, A_2), (A_3, A_4))$ and more generally an n-optic will have objects are 2^n tuples of objects in the base category. However, if we obtain an optic by repeated application of an optic functor, several of these objects will be the same and we will generally have $1+n$ different objects of the base category in play. E.g. $\mathbb{C}\mathbb{C}f$ for some morphism $f : A \to B$, would give a 2-optic $((A, A'), (A', A'')) \to ((B, B'), (B', B''))$.

For completeness, let us repeat this diagram, making use of the notation we established above for optic obtained from optic functors:

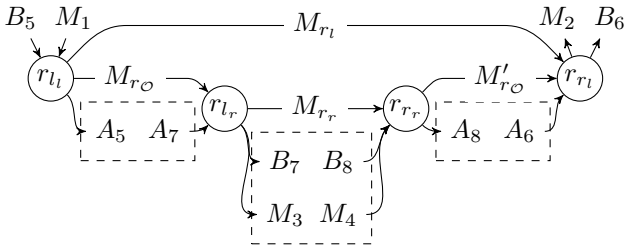


(21)

For completeness, let us proceed for one more step and explicitly draw a 3-optic. As before, we have a pair of two 2-optics:



(22)



(23)

Now, writing down the 3-optic is simply a matter of matching up the Ms, and setting corresponding gaps opposite each other. This is done in figure 2. Additionally, the composition rule for 3 optics is shown in the appendix in figure 4.

C. Well-behaved primitives

We say a primitive of an optic functor € is *well behaved* if the LHS of $€f$ is simply the original function 'f', possibly with a capture of the input or output value, i.e. $€f = \langle \Delta_{SSA} \fatsemi (id \otimes f) | f^\star \rangle$ or $€f = \langle f \fatsemi \Delta_{SSA} | f^\star \rangle$. For functions of this form, various higher order concepts simplify and we can perform analysis efficiently using dynamic programming. Speciafically, for an order n transform, we only need to do $n^2 - 1$ work rather than 2^n as would naively be expected. To see this, note that by repeatedly applying the transform, we can commute any function in our n-optic to just be a structural transform together with $f^{\star\star\cdots}$ where the number of stars is the total number of stars in the original expression. Then, letting numbers indicate the number of stars, we get the following sequence of patterns for increasing optics: 0, 01, 0121, 01212321, 0121232123432321, which is amenable to dynamic programming analysis.

Is this actually a useful notion? Certainly more advanced. Let's leave it here for now.

VI. Reverse mode AD

There are many ways to describe the difference between forward and reverse mode AD, but using our newfound language, it becomes quite simple: Forward mode AD pushes forward tangent vectors, while reverse mode AD pulls back cotangent vectors.

Now for the payoff from all of our hard work from the previous sections: Computing pullbacks of differential forms has the structure of an optic. As a concrete example, suppose we wanted to compute the gradient ∇f. Tracing through our definitions above, we find that:

$$\nabla f(x) = f^\star(dx_1(f(x))) \tag{24}$$

so two evaluate this function we first have a covariant evaluation of f, followed by a contravariant evaluation of $f^\star$. Now for the formal construction: Let $\mathcal{C}$ be the category of Riemannian manifolds. We have an optic category $\mathbf{Optic}_{\mathcal{C}}(T^\star M)$ [4] that pairs to each manifold its tangent bundle. Now, guided by the above, we defined a functor:

$$\overleftarrow{\partial} f : \mathcal{C} \to \mathbf{Optic}_{\mathcal{C}}(T^\star M) \tag{25}$$

where as indicated, $\overleftarrow{\partial} M = T^\star M$ and on morphisms $\overleftarrow{\partial} f = \langle (a \to (a, f(a)), (a, z) \to f^\star|_a(z) \rangle$.

We call this functor the **Reverse mode AD optic functor**.

There is very little left to do at this point, but our program above did teach us that we need to define a few things:

For copy, we simply write

$$\overleftarrow{\partial}\, copy(x) = (x, x), (\Delta_1, \Delta_2) \to \Delta_1 + \Delta_2 \tag{26}$$

which is the unique map that preserves linear of the pullback. For delete:

$$\overleftarrow{\partial}\, delete(x) = () \to 0 \tag{27}$$

For branch we have a choice. We could either inherit the behavior from delete or return a poison value on the backwards pass. This is a user level policy decision.

Lastly, we of course need to define derivatives for our primitives. We defer this task to ChainRules.jl.

VII. Higher order AD

A. Straightforwardly

Let us investigate the structure of higher order AD. One straightforward way to obtain the higher order transform is to simply treat the encoded optic as yet another SSA program. As an example, we may explicitly do the transform $\overleftarrow{\partial}\overleftarrow{\partial}$sin:

Snippet 14

```
function ∂̵←∂̵←sin(x)
    sin(x), Δ->begin
        Δ * cos(x), Δ′->begin
            Δ′*Δ*(-sin(x)), Δ′*cos(x)
        end
    end, (Δ‴,Δ″)->begin
        Δ″*cos(x) + Δ‴
```

[4] We're cheating here a little bit by calling the pair (A, A') simply $T^\star M$, technically we have a dependent optic $(x : M, T_x M)$, but since our programming language does not care about such things, we're happy to cheat an let others work out the tedious details.

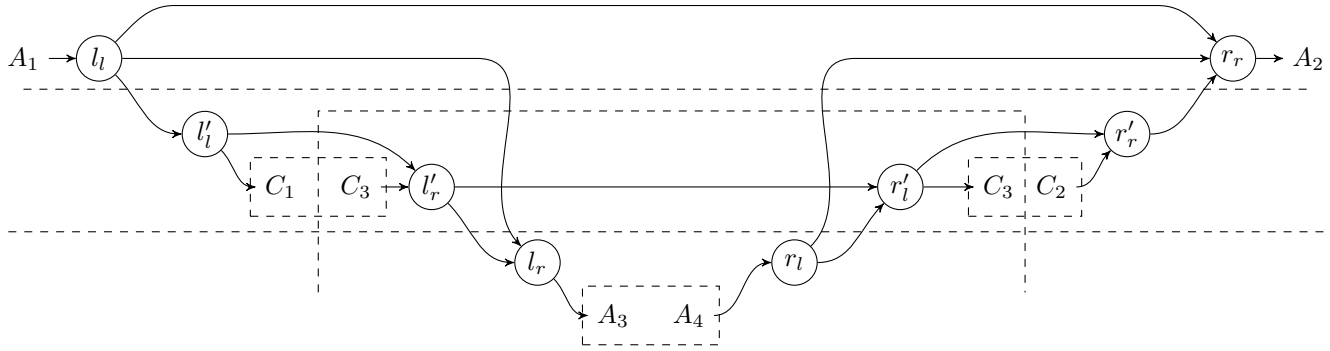


Fig. 1. Composition of two second order optics layed out horizontally. Indicated by dashed lines are the cut/glue lines of the original optic (as well as the resulting optic, which corresponds with the cutline of the second optic).

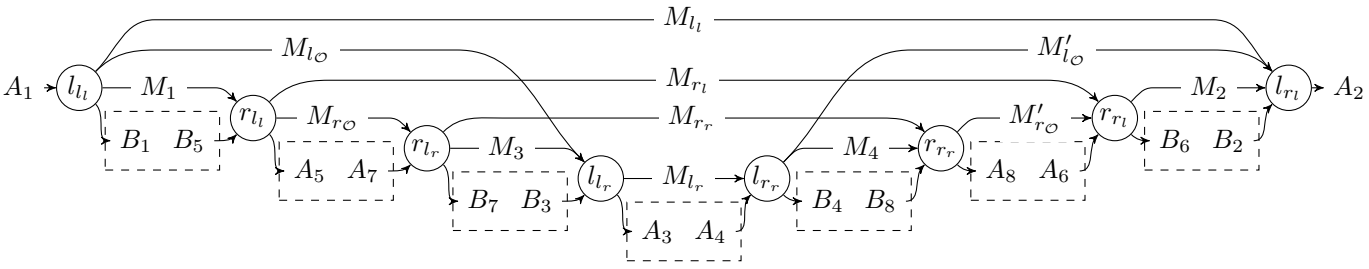
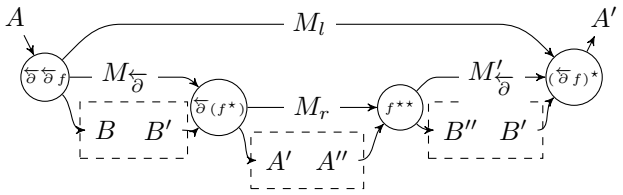


Fig. 2. Wiring diagram for the canonical 3-optic $(A_{1-8}) \nrightarrow (B_{1-8})$

```
    end
end
```

Recall from above the diagram for an optic functor

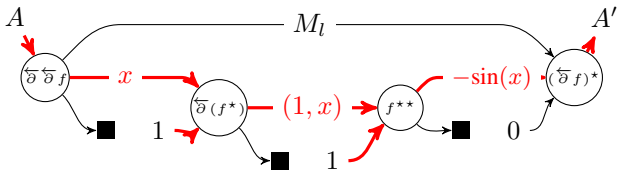


(28)

We can identify the various arrows with concrete values in the program:

Diagram	Program
M_l	Capture of x in the last closure
$M_{\overleftarrow{\partial}}$	Capture of x in the first closure
B	Return value of $\overleftarrow{\partial}\,\overleftarrow{\partial}\sin$
B'	Δ
M_r	Capture of Δ, x by the inner closure
A'	Primal return value of the first closure $(\Delta * \cos(x))$
A''	Δ'
B''	Second return, inner closure $(\Delta' * \cos(x))$
$M'_{\overleftarrow{\partial}}$	First return, inner closure $(-\Delta' * \Delta * \sin(x))$
	Also, Δ'''
B'	Δ''

In particular, if we are taking the second derivative of sin, here is the dataflow path of the final answer:



(29)

For a more complex example, the same transform is done for a third order transform in the Appendix (figure 6).

B. Alternative encoding

The previous approach works, but it is somewhat unsatisfying. Why? For me the two primary reasons are that there is no manifest difference between the residuals and the "holes" of the optic and that control flow is somewhat non-transparent. At first order, the residual was encoded as the capture of the original closure. However, at second order, while most residuals are still closure captures, Δ'' is a residual, but does not look like a capture. This shows up more at third order. To see what I mean when I say control-flow is non-transparent, consider the order in which the functions are called at third order:

$$\overleftarrow{\partial}\,\overleftarrow{\partial}\,f, \overleftarrow{\partial}\,\overleftarrow{\partial}\,(f^\star), \overleftarrow{\partial}\,(f^{\star\star}), \overleftarrow{\partial}\left(\left(\overleftarrow{\partial}\,f\right)^\star\right), \left(\overleftarrow{\partial}\,f\right)^{\star\star}, f^{\star\star\star}, \left(\overleftarrow{\partial}\,(f^\star)\right)^\star, \left(\overleftarrow{\partial}\,\overleftarrow{\partial}\,f\right)^\star \tag{30}$$

(if it's not clear how to come up with this order, simply look at the second order diagram, add $\overleftarrow{\partial}$ in front of everything, then reverse the diagram, add $\star$ after everything

and append the reversed diagram to the one that had $\overleftarrow{\partial}$ added). Comparing this to figure 6 in the appendix, we see that in source code order, the sequence is $1, 2, 3, 6, 7, 5, 8,$ so we need to do a lot of jumping around to keep track of what's happening. Can we re-arrange the encoding to make it easier to understand what's going on? Let's take a clue from the diagrams we've been working with, which do make the control flow order manifestly clear. What if instead of our second order example above, we wrote it as:

Snippet 15

```
function ∂²sin(x)
    sin(x), Δ->begin
        Δ * cos(x), Δ′->begin
            Δ′*cos(x), let Δ‴ = Δ′*Δ*(-sin(x))
                Δ″->Δ″*cos(x) + Δ‴
            end
        end
    end
end
```

This avoids the two problems identified above: The Δ''' becomes an explicit capture and the order of operations becomes quite clear (outer functions are evaluated before their inner closures). This representation is a bit of a re-think of the standard transform, but it is entirely equivalent. To get more familiar with it, let's say, we'd like to take the third order transform of the following functions:

Snippet 16

```
function f(x)
    y = g(x)
    z = h(y)
    return z
end
```

We would have:

Snippet 17

```
function ←∂³f(x)
  y, ȳ = ←∂³g(x)
  z, z̄ = ←∂³h(y)
  z, Δ->begin
    α, ᾱ = z̄(Δ)
    β, β̄ = ȳ(α)
    β, Δ′->begin
      γ, γ̄ = β̄(Δ′)
      δ, δ̄ = ᾱ(γ)
      δ, Δ″->begin
        ϵ, ϵ̄ = δ̄(Δ″)
        ζ, ζ̄ = γ̄(ϵ)
        ζ, Δ‴->begin
          η, η̄ = ζ̄(Δ‴)
          θ, θ̄ = ϵ̄(η)
          θ, Δ⁴->begin
            ι, ῑ = θ̄(Δ⁴)
            κ, κ̄ = η̄(ι)
            κ, Δ⁵->begin
              λ, λ̄ = κ̄(Δ⁵)
              μ, μ̄ = ῑ(λ)
              μ, Δ⁶->begin
                ν, ν̄ = μ̄(Δ⁶)
                ξ, ξ̄ = λ̄(ν)
                ξ, Δ⁷->begin
                  o = ξ̄(Δ⁷)
                  π = ν̄(o)
                  return π
                end
              end
            end
          end
        end
      end
    end
  end
end
```

We see still have the same number of functions, so the complexity isn't reduced, but at least it's fairly orderly. We do have to take special care of primitives, because we're no longer just applying the same transformation to the rule function itself. So suppose we have some primitive rule like:

Snippet 18

```
struct ∇a
    x
end
(::∇a) = ...

function rrule(::typeof(a), x)
    b(x), ∇a(x)
end
```

For the first order, we would just expose the raw pullback as we do with the naive transform. However, for second order, things become slightly more complicated:

Snippet 19

```
function ←∂²a(x)
    (y, ȳ), ȳ̄ = ←∂rrule(a, x)
    y, Δ->begin
        (α₁, α₂), ᾱ = ←∂typeof(ȳ)(ȳ, Δ)
        (α₁, α₂), (Δ′₁, Δ′₂)->begin
            (Δ‴, β) = ᾱ((Δ′₁, Δ′₂))
            β, Δ″->begin
                # Drop gradient w.r.t. `rrule`
                (_, a', x') = ȳ̄(Δ″, Δ‴)
                return (a', x')
            end
        end
    end
end
```

It's worth comparing this with the explicitly written out and inlined version of sin above (which is a primitive).

It's also worth taking a look at the third order version of that example to see the generalizing pattern:

Snippet 20

```
function ←∂³a(x)
    (y, ȳ), ȳ̄ = ←∂²rrule(a, x)
    y, Δ->begin
        ((α₁, α₂), ᾱ) = ←∂²typeof(ȳ)(ȳ, Δ)
        (α₁, α₂), (Δ′₁, Δ′₂)->begin
            (Δ‴, β), β̄ = ᾱ((Δ′₁, Δ′₂))
            β, Δ″->begin
                (_, γ₁, γ₂), γ̄ = ȳ̄(Δ″, Δ‴)
                (γ₁, γ₂), (Δ⁴₁,Δ⁴₂)->begin
                    (δ₁, δ₂), δ̄ = γ̄((0, Δ⁴₁, Δ⁴₂))
                    δ₁, Δ⁵->begin
                        (ϵ₁, ϵ₂), ϵ̄ = β̄(δ₂, Δ⁵)
                        (ϵ₁, ϵ₂), (Δ⁶₁,Δ⁶₂)->begin
                            (ζ₁, ζ₂) = ϵ̄((Δ⁶₁,Δ⁶₂))
                            ζ₂, Δ⁷->begin
                                return δ̄(Δ⁷, ζ₁)
                            end
                        end
                    end
                end
            end
        end
    end
end
```

If this structure seems familiar, it should! In our 3-optic

diagram we build 3-optics as pairs of 2-optics that composed in a canonical way. This is just the very same thing (compare functions l_{x_x} with closures returned from $\overleftarrow{\partial}^2$rrule and r_{x_x} with closures returned from $\overleftarrow{\partial}^2$typeof($\overline{y}$)).

There is one additional case that's worth considering: We allow explicitly calling the derivative operator in the primal. What happens when we encounter such a function. The answer is fairly straightforward. We're basically doing the inverse of the rrule case above. In particular, we're adapting an $n+1$ optic to an n optic by appropriate interleaving. Suppose we have a primal function ∇ given by:

Snippet 21

```
function ∇(f, x)
    (a, ā) = ←∂(f, x)
    y = g(a)
    z = ā(y)
    return z
end
```

Applying our transform, we get:

Snippet 22

```
function ←∂∇(f, x)
    (a, ā) = ←∂²(f, x)
    y, ȳ = ←∂g(a)
    z, z̄ = ā(y)
    z, Δ->begin
        α, ᾱ = z̄(Δ)
        β = ȳ(α)
        ϵ = ᾱ(β)
        return ϵ
    end
end
```

Note that we went through all four callbacks of the 2-optic $\overleftarrow{\partial}^2(f,x)$, even though the top level optic only had two callbacks. For completeness, here is the third order version (second order transform):

Snippet 23

```
function ←∂²∇(f, x)
    (a, ā) = ←∂³(f, x)
    y, ȳ = ←∂g(a)
    z, z̄ = ā(y)
    z, Δ->begin
        α, ᾱ = z̄(Δ)
        β, β̄ = ȳ(α)
        ϵ, β̄ = ᾱ(β)
        ϵ, Δ′->begin
            λ, λ̄ = ϵ̄(Δ′)
            ζ, ζ̄ = β̄(λ)
            μ, μ̄ = λ̄(ζ)
            μ, Δ″->begin
                θ, θ̄ = μ̄(Δ″)
                ι = ζ̄(θ)
                κ = θ̄(ι)
                return κ
            end
        end
    end
end
```

VIII. Implementation details

The key to a performant reverse mode AD implementation is to pick efficient residual representatives M for our optic as well as introducing minimal additional overhead in the tensor. In julia, the latter means being amenable to type inference (as well as ideally not being significantly harder on type inference than a simple primal inference to avoid unexpected performance cliffs). So, how do perform type inference on morphisms from the optic? Well, if we pick a particular representative, we obtain morphisms in the underlying category that we can do with whatever we want, including performing inference. However, picking a representative of course fixes the residual, which we'd like to avoid. Additionally, we run into problems with nested application of the functor.

IX. Appendix

Printing the appendix is encouraged to use as a template for working through examples.

References

[1] M. Riley, "Categories of optics," 2018. [Online]. Available: https://arxiv.org/abs/1809.00738

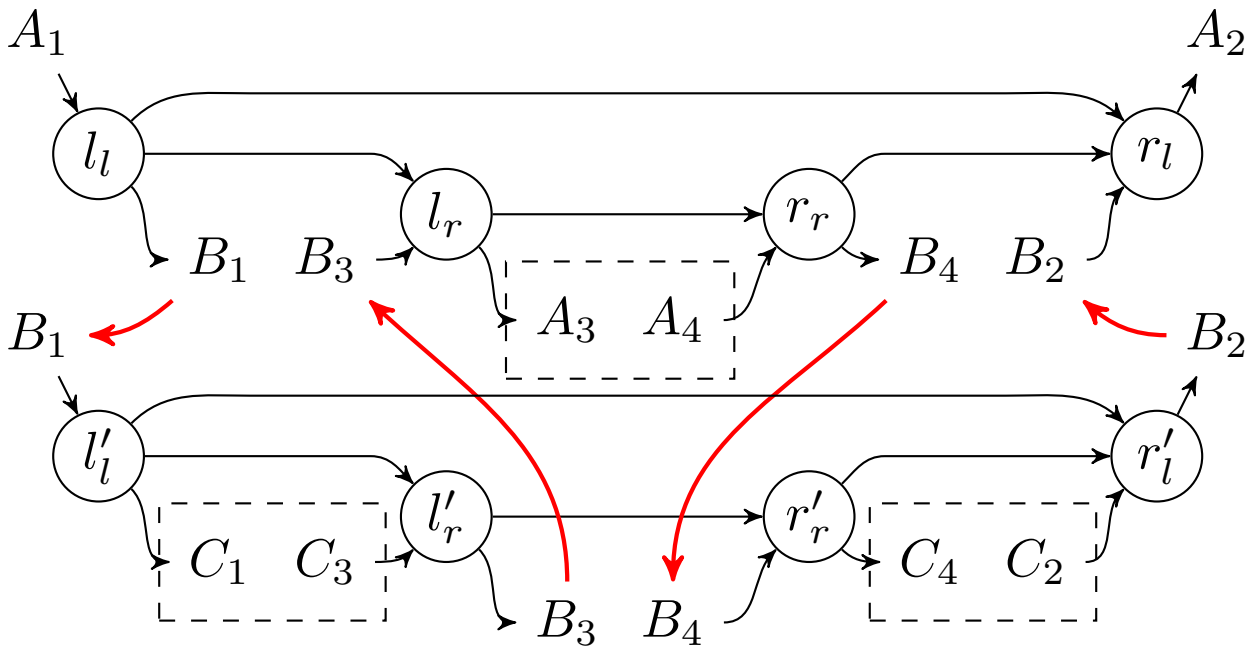


Fig. 3. Composition rule for 2-optics (larger version)

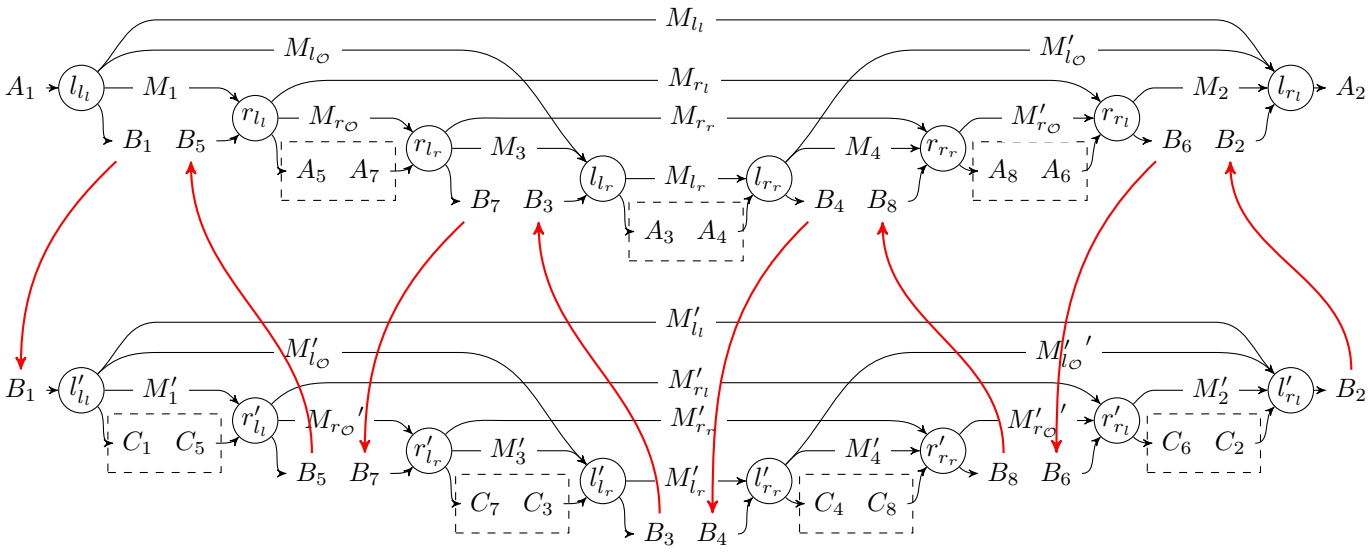


Fig. 4. Composition rule for 3-optics

```
function ←∂ ←∂ sin(x)
    sin(x), Δ->begin
        Δ * cos(x), Δ′->begin
            Δ′*Δ*(-sin(x)), Δ′*cos(x)
        end
    end, (Δ‴,Δ″)->begin
        Δ″*cos(x) + Δ‴
    end
end
```

Fig. 5. Explicitly written 2-nd order example (larger version)

```
function ←∂ ←∂ ←∂ sin(x)
    (sin(x), Δ->begin # ←∂ ←∂ (f*)
        Δ * cos(x), Δ′->begin # ←∂ (f**)
            Δ′*Δ*(-sin(x)), Δ′*cos(x), ((Δ⁴,Δ⁵),)->begin # f***
                (-Δ⁴*Δ′*Δ*cos(x) +
                    -Δ⁵*Δ′*sin(x),
                    -Δ⁴*Δ′*sin(x)),
                -Δ⁴*Δ*sin(x)+Δ⁵*cos(x)
            end
        end, (Δ⁶,(Δ⁷,Δ⁸))->begin # (←∂ (f*))*
            (Δ⁶*Δ*(-sin(x)) + Δ⁷,
                Δ⁶*cos(x) + Δ⁸)
        end
    end, (Δ‴,Δ″)->begin # ←∂ ((←∂ f)*)
        Δ″*cos(x) + Δ‴, Δ⁹->begin # (←∂ f)**
            (-Δ⁹*Δ″*sin(x)),
                (Δ⁹, Δ⁹ * cos(x))
        end
    end), (((Δ¹⁰,Δ¹¹),Δ¹²),)->begin # (←∂ ←∂ f)*
        Δ¹⁰*cos(x) + Δ¹¹ + Δ¹²
    end
end
```

Fig. 6. Explicitly written out third-order example