AD on Higher Order Functions

Sam Ritchie, Gerald Jay Sussman

2020-02-09

Contents

Gerry and I spent a good amount of time thinking through the behavior of the `D` and `j*` operators presented in Manzyuk et al. 2019, and the mists of confusion have mostly cleared.

We came up with an alternative way to fix Alexey's Amazing Bug; the behavior is potentially useful, certainly interesting, but not worthy of the name '`D`'. I've implemented it (and everything else we propose below) in the attached `implementation_updated.ss` file.

We also found a performance improvement that only invokes the tag replacement machinery from Manzyuk et al. when the user is in 'Alexey Territory'. Jeff, Barak, this might offer a path toward stronger complexity guarantees for the approach in your paper.

Finally, I've documented my remaining confusion about the behavior of `(j* f)` when functions are allowed into the domain. I'd love to include this feature in my SICMUtils port of GJS's library, so I spent a long time trying to understand how to document what it's doing. I'm becoming convinced that the behavior of the implementation presented strays from the differential geometric roots it was inspired by. It might be too weird to offer to users. But I'd love to be set straight here, so I've presented the case in detail in section 3.

Hopefully this is interesting and helpful! On we go.

1 Goals for this document

- lay out a spec for `D` as we understand it for known cases
- describe the menu for `(D f)` where f has a range of "function"
- document our alternative idea for fixing Alexey's Amazing Bug

- note the performance improvements from `epsilon-active?`
- Document the strange behavior of `(j* f)` with functions in the domain

2 The Spec for D

What is the contract offered by the `D` operator? The case is clear for functions whose range and domain are either real numbers or aggregates of real numbers. Our goal is to clarify the contract offered by `D` when applied to higher order functions — i.e., a function whose domain or range include functions.

A guiding principle here is that any behavior offered by an operator called `D` should behave like `D` from the mathematical literature.

2.1 `D` on Functions of Reals and Aggregates

`D` applied to a function `f` from $\mathbb{R} \to \mathbb{R}$ returns a new function `f'` that maps its input `x` to a unit increment in the value of `f` at `x`.

(Said another way, multiplying an incremental input `dx` by the quantity `((D f) x)` produces a proportional increment in the value of `f` at `x`.)

`D` applied to a function `g` with aggregate inputs and/or outputs — for example, $g : \mathbb{R}^n \to \mathbb{R}^m$ — satisfies the same contract. `((D g) x)` for some contravariant vector (`up` structure) `x` returns the Jacobian `J` of `g` at `x`. Given some unit-magnitude structural increment `dx`, The Jacobian-vector product `(* J dx)` is equivalent to a unit increment in the output of `g` in the direction of `dx` at the `n`-dimensional input `x`.

`D` applied to a function `h` of $n > 1$ arguments treats `h` as a function of a single `n` dimensional argument and returns the corresponding Jacobian.

2.2 (D f) with Function Range

Manzyuk et al. 2019 extends `D` to functions `f` of type $\mathbb{R}^n \to \alpha$, where

$$\alpha ::= \mathbb{R}^m \mid \alpha_1 \to \alpha_2 \tag{1}$$

By viewing

- `f` as a (maybe curried) multivariable function that *eventually* must produce an $\mathbb{R}^m$

- The derivative `(D f)` as the partial derivative of `f` with respect to the first argument

A 3-level nest of functions will respond to `D` just like the flattened, non-higher-order version would respond to `(partial 0)`. In other words, these two forms evaluate to equivalent results:

```
(let ((f (lambda (x)
           (lambda (y)
             (lambda (z)
               (* x y z))))))
  ((((D f) 'x) 'y) 'z))
;;=> (* y z)

(((partial 0) *) 'x 'y 'z)
;;=> (* y z)
```

If you keep calling the functional return values of `((D f) x)` until you get a non-function, the result should match the result of `(((partial 0) f) x ...)` if you passed all the arguments in at once.

2.3 Manzyuk and Nested Functions

There is a subtle difference between the behavior of a function `(f x)` and its derivative `((D f) x)`, using the Manzyuk et al. implementation.

If `(f x)` returns a function, then that function closes over a reference to `x`. If you engineer a program (see 2.4, coming up) where:

- `(f x)` takes another function, which then receives the closed-over `x` as an argument;
- you pass this `(f x)` to itself, so both closed-over `x` instances are passed into a binary function internally,

Then of course your program will make no distinction between instances of `x`. They are equal by both reference and value equality.

However! `((D f) x)` returns a function which, when you eventually provide all arguments, will return the sensitivity of `f` to the first argument `x`.

If you perform the trick above — pass `((D f) x)` into itself, and the `x` instances meet — should the final return value treat each `x` as the *same* instance?

Manzyuk et al says *no!*. If `((D f) x)` returns a function, that function closes over:

- the value of `x`
- an *intention* to start the derivative-taking process on that isolated copy of `x` once the final argument is supplied.

This distinct makes it possible to change the behavior of `((D f) x)` in potentially confusing ways through a refactor that makes no change to `(f x)` at any point.

2.4 Example: Alexey's Amazing Bug

Here's an example, written in the familiar setting of Alexey's Amazing Bug. (4.1 presents the GJS version of the original example I sent to the group.)

`arg-shift` takes some `g: R -> R` and returns a new function that acts like `g`, but adds `offset` to its input `a` before calling `g`:

```
;; R -> ((R -> R) -> (R -> R))
(define (arg-shift offset)
  (lambda (g)
    (lambda (a) (g (d+ a offset)))))
```

The multivariable equivalent is:

```
;; (R -> (R -> R) -> R) -> R
(define (arg-shift-multi offset g a)
  (g (d+ a offset)))
```

Treat this as a curried multivariable function by providing all arguments, and receive the expected `(exp 8)`:

```
((((d arg-shift) 3) dexp) 5)
;;=> (exp 8)
```

Now force a situation where both the `(R -> R)` argument and the function receiving it *both* come from the same call to `((d arg-shift) 3)`, and therefore both have a partial derivative "pending" from the initial `3` argument:

```
;; (R -> R) -> (R -> R), derivative pending from 3
(define f-hat ((d arg-shift) 3))

;; (R -> R), derivative pending from 3
```

```
(define f-arg (f-hat dexp))

((f-hat f-arg) 5)
;;=> (exp 11) in Manzyuk
```

The result is `(exp 11)` because each "derivative pending" from 3 triggers a different derivative-taking process when the final `5` is supplied.

Manzyuk et al. provides a referentially transparent implementation of `D`. As a consequence, the above result is identical to the form below, with all `define` bindings replaced by their values:

```
((((d arg-shift) 3)
  (((d arg-shift) 3) dexp))
 5)
;;=> (exp 11)
```

Referential transparency requires that each `3` trigger a different derivative computation.

2.5 What's the Problem?

This is a totally reasonable definition of the derivative. But I (Sam) found it to be confusing, which led to some interesting experiments by me and GJS that are worth at least writing down.

The trigger was a refactor that didn't change the final non-function *value* at any value of the `(offset, dexp, a)` arguments, but did change the result of the derivative.

Modify `arg-shift` by making the second argument a continuation that receives the rest of the expression and returns the final result:

```
;; R -> (((R -> R) -> (R -> R)) -> R)
(define (arg-shift-cont offset)
  (lambda (cont)
    (cont
     (lambda (g)
       (lambda (a) (g (d+ a offset)))))))
```

This version of `arg-shift` is now equivalent to a curried two-argument function. `((D arg-shift-cont) offset)` takes a function `cont` which receives a value identical to `f-hat`.

The difference is, because `cont` is evaluated *inside* the body of the new `arg-shift-cont`, it can use its `f-hat` multiple times and the augmented `x` values will contribute identically to the returned derivative.

```
;; ((R -> R) -> (R -> R)) -> R
(define (cont f-hat)
  ((f-hat (f-hat dexp)) 5))

(((d arg-shift-cont) 3) cont)
;;=> (* 2 (exp 11))
```

The result is now `(* 2 (exp 11))`. Because `f-hat` was used twice *inside* the body of the function returned by `((d arg-shift-cont) 3)`:

- both copies of `f-hat` started their derivative-taking process separately, using separate tags internally.
- the act of *crossing the right paren* forced these two derivatives back together.

This second item was the bug in my first email to the group. It manifested, if you'll recall, as an un-stripped dual number getting returned (or two clashing tags in the tag set, in the `scmutils` case).

What did I expect to happen? I wanted both implementations to return `(* 2 (exp 11))`, because I thought the refactor shouldn't change the result. I see now why this would be confusing and incorrect behavior for `D`, given the contract offered in 2.2.

`((D arg-shift) offset)` returns a function of `g` and `a` that, when called, should return the derivative of `g` at `(+ a offset)`. It should *not* matter how you calculate `a`!

In particular, If `a` happens to come from applying the same instance of `((D arg-shift) offset)` to some different pair, like `h` and `b`, the offsets had better not be treated like the same variable. This is, of course, the purpose of all of the tag-replacement machinery in Manzyuk et al.

2.6 Is there another way?

The original (Alexey's Amazing) bug shows up in equation `(12m)` of Manzyuk et al.:

$$\mathrm{tg}\,\epsilon_0\left(\mathrm{tg}\,\epsilon_0\left(h(y)+2h'(y)\epsilon_0\right)\right) \tag{2}$$

both `f-hat` instances attempted to extract the same epsilon, and the outer instance found nothing and returned `0`. The tag replacement machinery in the paper both solves this problem and prevents different nested `f-hat` calls from confusing their perturbations.

There is a different way to fix the double-extraction bug:

- Break referential transparency and make the user start the derivative-taking process with an explicit side effect — a call to the function returned by `(D f)`.
- If `((D f) x)` returns a function, let the closed-over, augmented `x` instances "cross-talk" in nested examples;
- Keep an explicit stack of in-progress tags;
- If a derivative-taking function sees (during execution) that its tag is already on the stack, pass its result back up un-extracted.

This approach would behave identically for all higher-order-function examples that don't nest. In contrast to Manzyuk et al., however, both examples presented in 2.4 would return `(* 2 (exp 11))`.

The implementation exposed two efficiency improvements to the tag replacement solution for higher-order functions, in the case where the returned function is *not* called in a nested context.

First, I'll describe what the new results would "mean", then how to implement this behavior.

2.6.1 Semantics, Meaning

The semantics above would let you measure the sensitivity of numbers you produce from an arbitrary execution graph to some explicitly tagged input.

As an example:

```
(let* ((offset 3)
       (f-hat ((d arg-shift) offset)))
  (list
   ((f-hat dexp) 5)
   ((f-hat (f-hat dexp)) 5)
```

```
   ((f-hat (f-hat (f-hat dexp))) 5)))
;;=> ((exp 5) (* 2 (exp 11)) (* 3 (exp 14)))
```

If the augmented `3` could interact across `f-hat` copies, each of the entries in the list would respond with a larger-by-one constant factor to an incremental change in offset.

If you want the sensitivity of the whole `((f-hat (f-hat exp) 5)` computation to the single knob attached to "3" — if you want to calculate an increment of the whole expression proportional to an increment in that single "3" — the `(* 2 (exp 11))` is the right answer, since an increment will affect both nested calls together. This seems like a reasonable tool to want.

If instead you are looking for an increment in the `((f-hat ....)  5)` given an increment in its "3" value (independent of whatever you pass in for `...`), then `(exp 11)` is correct. Wiggling the outer "3", if the inner `(f-hat exp)`'s value stays pinned at "3", should give you `(exp 11)` sensitivity.

This implementation would *not* be referentially transparent, because different calls to `(d arg-shift)` start separate derivative-taking processes. This is called out as an explicit problem in equation 15 of Manzyuk et al.

But after the first argument is passed to `(D f)`, the user is free to structure their program (move the parentheses) in any way they like. They'll always get the same result.

If your goal is to track the program's sensitivity to changes in the argument to `(D arg-shift)`, then you need the ability to explicitly distinguish calls:

```
(let* ((offset 3)
       (f-hat1 ((d arg-shift) offset))
       (f-hat2 ((d arg-shift) offset)))
  (list
   ((f-hat1 dexp) 5)
   ;; two separate f-hat instances
   ((f-hat2 (f-hat1 dexp)) 5)

   ;; one f-hat1, two f-hat2 gives a 2x factor
   ((f-hat2 (f-hat2 (f-hat1 dexp))) 5)))
;;=> ((exp 5) (exp 11) (* 2 (exp 14)))
```

You might read this result as the derivative of a sub-graph of a program you're defining on the fly, with respect to the functional node you passed to `D`. (Is "Calculus on Graphs" an existing field?)

In more detail: If `(f x)` returns a function, think of `f` a source in the program's execution graph.

Function return values are new nodes in this graph, and non-function return values are sinks. With this view:

- `((D f) x)` is still the partial derivative with respect to some input in an $\mathbb{R}^n \to \mathbb{R}^m$ function, and `x` is the first of the $\mathbb{R}^n$ inputs
- The other `n-1` inputs come from each call to the function `((D f) x)`, or any function that `(((D f) x) x2)` returns, etc.
- each of the `m` outputs comes from calling one of `m` functions that branched out from the original `((D f) x)`.

You have a vector-valued function, but you are getting each of the `m` results from separate functions, and therefore at different places in your program.

If I convert some piece of my program to continuation-passing style, `((d current-continuation) x)` produces the sensitivity of the rest of my program to `x` in both implementations. `*fix-three?*` allows us to sample sensitivity to `x` without this transformation.

Maybe this is a stretch.

2.6.2 Implementation

I've implemented the above alternative Alexey's Bug fix in `implementation_updated.ss`. Turn it on by setting the flag `*fix-three?*` to `#t`.

The key idea is to maintain a stack of `*active-epsilons*`. Each time a derivative-taking higher-order function is applied to fresh arguments, it does this via `with-active-epsilon`. Any nested call looking to extract the same `epsilon` will see `(epsilon-active? epsilon)` return `#t`.

```
(define *active-epsilons* '())

(define (epsilon-active? epsilon)
  (memq epsilon *active-epsilons*))

(define (with-active-epsilon epsilon f arg)
  (let ((old *active-epsilons*))
    (set! *active-epsilons* (cons epsilon *active-epsilons*))
    (let ((result (f arg)))
```

```
      (set! *active-epsilons* old)
      result)))
```

Any function returned by `tg` now checks for `(epsilon-active? epsilon)` before calling `tg` internally. If its `epsilon` is active, it passes its value up without calling `tg`, leaving it intact for the waiting higher-level call.

This actually gives a nice savings in the `prim` implementation, spotted by GJS. You only need to perform a tag substitution if some function above you is waiting for the same tag. Otherwise, instances of your tag can't get in via your arguments.

I've implemented these savings under a `*save-work?*` flag. Here's the change in `prim`:

```
(lambda (y)
        (if (epsilon-active? epsilon)
      (let ((epsilon2 (generate-epsilon)))
              (subst epsilon
                          epsilon2
                          (prim epsilon
                    (with-active-epsilon
                     x (subst epsilon2 epsilon y)))))

      ;; Easy and cheap!
      (prim epsilon (with-active-epsilon x y))))
```

And a similar change in `subst`:

```
(if (epsilon-active? epsilon2)
    (let ((epsilon3 (generate-epsilon)))
            (subst epsilon2 epsilon3
                        (subst epsilon1 epsilon2
                    (x (subst epsilon3 epsilon2 y)))))
    ;; Do the easy thing!
    (subst epsilon1 epsilon2 (x y)))
```

Perhaps this will allow us to tighten up the complexity guarantees of this patch to forward-mode AD. Rather than `(D f)` costing a constant factor, we might say that `(D f)` usually costs this, but in 'Alexey Territory' cost becomes some function of:

- dimension of the input structure to `f`

- nesting level

I'll leave this as an exercise for the reader :)

The only other change is `extract-fix-three`, similar to `tg` but only called by `d` to perform the final tangent extraction:

```
(define (extract-fix-three epsilon x)
  (cond ((dual-number? x)
         ;; If tg is attempting to extract an epsilon that a higher level is
         ;; waiting for, (tg epsilon x) acts as (identity x).
         (if (epsilon-active? epsilon)
             x
             (tg epsilon x)))

        ;; The returned procedure calls (x y) with epsilon marked as active.
        ;; Inside that (x y) call, the (epsilon-active? x) branch in the
        ;; (dual-number? x) case above will return true.
        ((procedure? x)
         (lambda (y)
           (extract-fix-three epsilon (with-active-epsilon epsilon x y))))

        ;; All other cases are identical to a call to tg.
        (else (tg epsilon x))))
```

If `*fix-three?*` is `#t`, `d` calls `extract-fix-three` instead of `tg` to extract its result:

```
(let ((epsilon (generate-epsilon)))
        (extract-fix-three epsilon (f (create-dual-number epsilon x 1))))
```

`implementation_updated.ss` has tests and usage examples of these new behaviors.

2.6.3 Request for Jeff, Barak's Comment

Jeff, Barak — is this a good tool to add to the toolbox? I agree that the semantics in the paper are the best default for `D`. But I'm sure you've thought about some operation with these semantics. Does this new behavior correspond to some pre-existing idea in the mathematical literature?

2.6.4 Middle Ground between Two Extremes

These are two extremes. You might also write a `fork` function that explicitly introduced the Manzyuk behavior into a function returned by `((D f) x)` in the `*fix-three?*` version. This would switch "share all instances" mode to "replace tag at every function boundary" mode.

Manzyuk implicitly calls `fork` every time a returned function is called. `*fix-three?*` never calls it. Some more enlightened library author might use `fork` to build a custom sensitivity-measuring function and then provide it to the user. This might recover the "curried derivative" idea with more flexibility.

You might also *warn* the user if:

- you're in `*fix-three?*` mode
- a function with tag `tag` gets called when `(epsilon-active? tag)` returns `#t`, signaling a nested call

This is the only case where behavior would be different. They could resolve the warning by:

- explicitly calling `fork` on the nested function, or by
- wrapping the computation in a form that sets the `*warn?*` variable to `#f` within its scope

2.7 Third Approach: Curried Directional Derivative

A third way to make sense of `D` on a higher-order function is to take inspiration from the `j*` approach and accumulate a directional derivative, with distinct basis directions specified for each argument in the curried version of a multivariable function.

As the discussion in section 3 will make clear, It seems that specifying directions in a directional derivative is the key API challenge for this whole enterprise.

2.8 Can we make the epsilon gensym completely explicit?

Can we get our referential transparency back by making tag assignment explicit, and defaulting to `gensym` generation?

I think this is a hard *no*, after much thought. I am convinced that you can't, in general, open up tag assignment to the user and still call the function `D`. There is almost nothing the user can *do* with the tag they've explicitly chosen, since the call to `extract` is hidden inside `D`.

The only valid way to use an explicit tag is to force distinct calls to `((D f) x)` to use the same tag for the same `x`.

You could come close to doing this automatically by memoizing the gensym call on the `x` argument as Jeff suggested in our email correspondence. But you can never memoize on the function's *value*, only the particular reference you have in hand:

```
(define (arg-shift offset)
  (lambda (g)
    (lambda (a) (g (d+ a offset)))))

(let* ((df (d arg-shift))
       (f-hat1 (df 3))

       ;; These two cases will result in `f-hat2` tracking a different or the
       ;; same tag (respectively) as `f-hat1`:
       (f-hat2 (df 3))
       (f-hat2 f-hat1)

       ;; If you memoize tag assignment on 'x==3', you'd always get the same
       ;; tag. But if each (d arg-shift) has its own memoization cache then
       ;; THESE two forms would act differently, pushing the referential
       ;; transparency problem back up one level:
       (f-hat2 (df            3))
       (f-hat2 ((d arg-shift) 3))

       ;; And as Jeff pointed out, you can't memoize on a function, since
       ;; function equality is undecidable.
       )
  ((f-hat1 (f-hat2 dexp)) 5))
```

What are some things that can go wrong? (All of these only apply to the `*fix-three?*` semantics; Manzyuk forces fresh tags at every level so it doesn't matter what you assign.)

2.8.1 Same tag, different values

```
(let ((f-hat1 ((D f) x 0))
      (f-hat2 ((D f) y 0))) ...)
```

In *distinct* argument positions, this technique gives you a directional derivative with respect to `x` and `y`, with a `1` in each direction. But I don't think there's any way to make sense of the results as a "derivative" if you cook up a situation like this, with two distinct values lifted into the same tangent space from the same argument position, and then allow the values to mingle with `((f-hat1 (f-hat2 exp)) 5)`.

It turns out that this is what happens with `j*` when you provide a function as argument.

2.8.2 Nested tag clashing

You might choose a tag already in play at a level above you. You could solve this by maintaining a global map of `{user-tag -> (fresh-tag)}`, so at least you'd never clash with a gensymmed tag.

2.8.3 Trouble with the Jacobian

The Jacobian calculation on a higher order function is a more complicated beast. The Jacobian is a structure like the input structure (of opposite variance), where each entry is the partial derivative with respect to the corresponding entry in the input structure.

If each entry is a function, and you:

- explicitly supply tags
- supply identical tags to different entries
- take the resulting structure of functions and tangle different entries

Then I don't know how to interpret the output-tangling. If you tangled entries that shared tags you would end up with a curried directional derivative of those entries. This feels like something to forbid!

2.9 What about Reverse Mode?

Reverse mode has the two same semantic extremes, for the same reasons. You can choose to employ the `subst` machinery to keep inputs separate, or allow them to cross-talk.

I do think that the `*active-epsilons*` stack will make it simple to tie-break between `tape-cell` and `dual-number`. The question of "do I put the dual into the tape cell, or vice versa?" is resolved by deciding which side's tag is currently in play. One way to decide is to force a global ordering of gensymmed tags (by using a timestamp, for example). Then the greater of the two tags is the one in play.

The `*active-epsilons*` stack makes it explicit which epsilon is associated with your current level of nesting. This makes it easier in mixed-mode AD to decide, when combining a dual number with a tape, which epsilon is 'greater'. `lift-real*real->real` in R6RS-AD provides a nice example of code that would become simpler with `*active-epsilons*`.

3 Directional Derivative via j*

After the above investigation of allowing functions into the range of `(D f)`, I turned my attention to the directional derivative operator `j*` in section 9 of Manzyuk et al.

My goal was to understand how `(j* f)` would behave if `f` took a function argument. I think that the resulting behavior might be just as strange as the behavior we'd see by allowing nested calls to mix their captured arguments in the style of 2.6.1.

`((j* f) g g')` augments `g` such that, whenever it is called inside `f`, it assigns a tangent vector of `(g' arg)` to its argument. The problem is that `g'` can't be a directional derivative like `(j* g)`, because `(j* g)` takes two arguments, but `f` as written calls `g` with one argument. So `g'` has to assign a default tangent vector. This is only possible for $\mathbb{R}$ inputs (assign 1) or function inputs (assign `(D arg)`).

But assigning a tangent vector of 1 on every call to `g'` is something like taking a directional derivative on a `n`-dimensional manifold, where `n` is the number of calls to `g`, and the direction is a structure with `n` "1" entries.

As an example:

```
(let ((f (lambda (g)
           (d* (g 1)
               (g 2)))))
  ((j* f) exp (d exp)))
;;=> (* 2 (exp 1) (exp 2))
```

the `g` passed into `f` is an augmented `g`, so both `1` and `2` are assigned a tangent vector of 1. The whole form returns `(* 2 (exp 1) (exp 2))`, the sum of partials with respect to the `1` and `2` arguments to `g`.

Here is the equivalent directional derivative call with an explicitly structured argument. My claim is that this form will produce the same result:

```
(let ((f (lambda (input)
           (let ((one (car input))
                 (two (cadr input)))
             (* (exp one)
                (exp two))))))
  ((j* f) (list 1 2) (list 1 1)))
;;=> (* 2 (exp 1) (exp 2))
```

Because `j*` is so careful to expose the tangent vector `x'` as an argument, this feels like a problem with the API on offer. Is there some differential geometry concept with this behavior?

The trouble stems from the attempt to avoid making the user specify a tangent vector, since that would require rewriting `f` to pass explicit tangent vectors in all calls to the augmented `g`.

Brace for a more detailed writeup, including a tour of section 9 and the consequences of function arguments to `(j* f)`.

3.0.1 Definitions

Following section 9:

- Given some manifold M, we can associate each point x on M with a tangent vector x'. x' is an element of the tangent space T_xM (subscripted since each `x` has its own tangent space).
- The ordered pair (x, x') is an element of the tangent bundle TM.
- Given manifolds M and N, and a function $f : M \to N$, the pushforward operator maps $(M \to N) \to (TM \to TN)$. So `(pushforward f)` maps elements of the tangent bundle of M to elements of N's tangent bundle.
- For a 1-dimensional manifold, the unit tangent vector is always equal to 1.

What is the type of the tangent vector of a function $f : M \to N$? Manzyuk et al. notes that the type $T(M \to N)$ is defined as $M \to TN$.

There are (of course) many functions one could choose. The ambiguity is in the choice of tangent vector for each point in M.

3.0.2 D Interface

Assuming the operator-overloading approach to forward-mode AD, any function `f` that we can pass to `D` on is both `f` and `(pushforward f)`.

That is because, thanks to generic functions, `f` can accept both:

- some input point $a : M$ on a manifold M
- a differential object $(a : M, a' : T_a M, tag)$, i.e., a (tagged) element of the tangent bundle of manifold M

Given $f : M \to N$, `D` is *not* the pushforward operator; `(D f)` returns a function of type $a : M \to T_{f(a)}N$, i.e., a function from an element of M to an element of the tangent space of N at the point $f(a)$.

To work with this interface, `(D f)` has to be able to assign a tangent vector to its argument. It does this by *only* allowing a single argument in $\mathbb{R}$ and always assigning the only possible unit tangent vector of `1`. This is a directional derivative, of course, but the direction is trivial, so `(D f)` can get away with no extra user argument.

To take a directional derivative at some point in a greater-than-1 dimensional manifold, the user has to break the ambiguity in direction by supplying an explicit structural tangent vector, `x'`. `j*` in Manzyuk et al. accepts a second argument to support this.

3.0.3 Functions in the domain of (j* f)

Consider the case where `(j* f)` accepts a function argument `g`.

Repeating a definition from above: the type of the tangent vector of a function $g : M \to N$ is $M \to TN$.

That conveniently matches the signature of `(D g)`. Given $g : M \to N$, we can use `g` and `(D g)` to make a function with the correct type:

```
(lambda (x) (bun epsilon (g x) ((D g) x)))
```

In fact, this is what Manzyuk et al. does in equation 35c. `sqr` and `(D sqr)` both get "bundled" into a function like the one in the line above:

$$\begin{aligned} \text{mapPair } fl &\triangleq (f(\text{ fst } l)), (f(\text{ snd } l)) \\ \text{sqr } x &\triangleq x \times x \\ \vec{J}\,\text{mapPair sqr } (D \text{ sqr })(5, 10) &\Longrightarrow (10, 20) \end{aligned} \tag{3}$$

Now! Here come dragons, for the reasons I described in section 3. This behavior is troubling to me for the following reasons:

1. `(D g)`, as I note above, takes a numeric input, and always assigns a tangent vector of `1`.
2. `((j* f) g (D g))` passes an augmented version of `g` to `f`; inside the body of `f`, any time this augmented `g` is called with a point in M, it maps that point to $TN\ldots$ but every input receives the *same* tag. (There is a tag substitution step, but I believe it's not doing anything. See 3.0.5.)
3. There is no way (or reason) to assign a different tag to each call to the augmented `f`, because which tag would the new function extract?

Repeating the example from the introduction, these two functions return the same result:

```
(let ((g (lambda (f)
           (d* (f 1)
               (f 2)))))
  ((j* g) exp (D exp)))
;;=> (* 2 (exp 1) (exp 2))

(let ((f (lambda (input)
           (let ((one (car input))
                 (two (cadr input)))
             (* (exp one)
                (exp two))))))
  ((j* f) (list 1 2) (list 1 1)))
;;=> (* 2 (exp 1) (exp 2))
```

And repeating a comment: assigning a tangent vector of 1 on every call to `g'` is something like taking a directional derivative on a `n`-dimensional manifold,

where `n` is the number of calls to `g`, and the direction is a structure with `n` 1-valued entries.

3.0.4 API Observation

I think most of my confusion has stemmed from our desire to be able to call `g` and `(j* g)` identically, without wiring tangent vectors into each argument spot. This is manageable with a single $\mathbb{R}$ input or an explicit tangent vector, but quite confusing when a directional derivative is built on the fly inside of a function.

But it might still be a great idea!

You could imagine, for example, a program transformation from this form:

```
(let ((g (lambda (f)
           (* (f 1)
              (f 2)))))
  ((j* g) exp (D exp)))
```

To some Jacobian-like thing that shows you the partials with respect to each call to `f`. In this imagined UI you could adjust the tangent vector contribution from each argument on the fly, or change the values and see how the derivative of the full form changes.

You would lose the property that the return value had a structure you could multiply by an increment in the "input" to get an increment in the output. There is no clear input structure.

You might also break the ambiguity by forcing the second argument to `j*` to be a pair of:

- a mapping of $M \to TN$
- a mapping of $m : M \to T_m M$, i.e., a function that gives you a tangent vector for each point in M. This is indeed what a vector field does in differential geometry.

This won't solve the problem above, as we're not interested in the tangent vector at each *value* — we want a tangent vector entry for each invocation of the augmented `g'`.

One more note before I conclude.

3.0.5 Does j* Tag Replacement Accomplish its Goal?

The implementation of `j*` in section 9 seems to want each invocation to act on a *distinct* tag. The paper performs a tag-replacement step before and after applying `x` and `x-prime`.

```
(lambda (y)
        (let ((epsilon2 (generate-epsilon)))
          (subst epsilon epsilon2
                 (bun epsilon
                              (x (subst epsilon2 epsilon y))
                              (x-prime (subst epsilon2 epsilon y))))))
```

I am fairly convinced that this is doing nothing. How can it do anything? Tag replacement makes sense with the `tg` and `subst` implementations for `D` because `tag` is extracted out before substituting `fresh` back in for `tag`. This implementation simply swaps then swaps back.

I believe that this implementation is equivalent:

```
(lambda (y)
        (bun epsilon (x y) (x-prime y)))
```

Is there an example that behaves differently for these two cases? I am probably wrong and confused here, but I can't see it.

3.0.6 Questions and Summary:

The directional derivative is the big idea living behind the `D` and `j*` operators. This suggests certain behavior the following cases for `(j* f)`:

- For $f : \mathbb{R} \to \mathbb{R}^m$, assign a default tangent vector of 1.
- If f returns a function, have the function returned by `(j* f)` optionally accept a second `x'` argument, but have it default to `0` for $\mathbb{R}$ inputs, `(zero-like x)` for structural inputs and a non-augmented `g` for some functional input `g`.
- For f with a structural first argument, force the user to provide an explicit tangent vector.
- Treat n total (curried) arguments and their tangent vectors as a single structural argument with a structural tangent vector built out of the supplied pieces.

These all feel nice for the user, because the user always supplies the tangent vector of an argument along with the argument itself (or a sane default is supplied).

The current `j*` semantics for a functional input, however, seem problematic. The function passed to `(j* f)` has to either:

- assign some default non-zero tangent vector to its argument every time it's called
- Force the user to provide an explicit tangent vector at each call site; this would require a rewrite of the function!
- Somehow arbitrarily choose one internal call to privilege, and assign a tangent vector of `1` in that case and `0` to all other calls.

Or... what? Is there some Jacobian-like thing we can return?

This presents a juicy API design question:

- Are there real-world problems where we would prefer that `(j* f)` take a functional argument, instead of suggesting that the user rewrite their function?
- If allowed, what is a sane default behavior?
- How do we sanely let the user customize?

4 Miscellany, Appendices

This section contains some notes that felt relevant to include but don't fit into the narrative above.

4.1 GJS's Cake Example

This email from GJS is a nice distillation of the initial example we considered together. In 2.4 I recreated the behavior in the more familiar terrain of the Amazing Bug, but the original example tugs my "what is right?" intuition in the opposite direction, and is worth recording. Enter GJS.

Hi Jeff,

Gee... I hope I can clarify my confusion. Perhaps you can help unconfuse me.

In the following code I will explain why I think there is a problem (thank you Sam for alerting us to this!).

I will not use any fancy features, in the first demonstration of the problem.

I will use Church pairs to avoid any data structure problems that might contaminate the argument.

```
;;; Church pairs

(define kons
  (lambda (a d)
    (lambda (m)
      (m a d))))

(define kar
  (lambda (x)
    (x (lambda (a d) a))))

(define kdr
  (lambda (x)
    (x (lambda (a d) d))))
```

I think that this is the original example that Sam brought up for us to consider. For some hysterical reason I am using the name `CAKE` for what he called `F`.

```
(define (cake x)
  (lambda (cont)
    (cont (lambda (y)
            (* x y))
          (lambda (g)
            (g x)))))
;Value: cake
```

The problem is that the following two computations produce different answers:

```
(((D cake) 5)
 (lambda (f1 f2)
   (f2 f1)))
;Value: 10

(let ((both (((D cake) 5) kons)))
```

```
  (let ((f1 (kar both))
        (f2 (kdr both)))
    (f2 f1)))
;Value: 1
```

I will explain why this is problematical.

It appears that we are doing the same thing in both of the above cases. Indeed calls to `CAKE` give the same value.

```
((cake 5)
 (lambda (f1 f2)
   (f2 f1)))
;Value: 25

(let ((both ((cake 5) kons)))
  (let ((f1 (kar both))
        (f2 (kdr both)))
    (f2 f1)))
;Value: 25
```

Now, using a more fancy feature (symbolic evaluation) I get:

```
(define (gcake x)
  (lambda (cont)
    (cont (lambda (y)
            ((literal-function 'f) x y))
          (lambda (g)
            (g x)))))

(simplify
 ((gcake 't)
  (lambda (f1 f2)
    (f2 f1))))
;Value: (f t t)

(let ((both ((gcake 't) kons)))
  (let ((f1 (kar both))
        (f2 (kdr both)))
    (f2 f1)))
;Value: (f t t)
```

So both uses of `GCAKE` are equivalent. But the derivatives are different.

```
(simplify
 (((D gcake) 't)
  (lambda (f1 f2)
    (f2 f1))))
;Value: (+ (((partial 0) f) t t) (((partial 1) f) t t))

(let ((both (((D gcake) 't) kons)))
  (let ((f1 (kar both))
        (f2 (kdr both)))
    (f2 f1)))
;Value: (((partial 1) ((partial 0) f)) t t)
```

I think that the first answer is correct, because it is what I would naively expect from the fact that we got `(f t t)` from the evaluation of the undifferentiated `GCAKE`.

Perhaps there is a reasonable explanation for this, but I don't see it clearly.

4.2 Alternative idea for (D f) with function range:

For a higher-order function `f`, `((D f) x)` returns a function which returns a function etc, until eventually you get a number. We interpret that as `(((partial 0) f) x ....)` of the equivalent multi-argument version, only because we have to pick a privileged direction if we don't want the user to have to provide one.

You might also allow the user to specify a tangent vector `x'` for each curried argument. The final return value would be a directional derivative, where each step supplies one component of the equivalent multi-argument tangent vector. This is the proper shape of the function-returning version of `j*`.

To recover the behavior of `D` on a curried 3-argument function, pass tangent vector 0 for the second and third arguments. note that these should be equivalent:

```
((((D f) x) y) z)
((((D f) x 1) y 0) z 0)
```

Or really, to allow structured remaining arguments:

```
((((D f) x 1) y (zero-like y)) z (zero-like z))
```

4.3 Multiplying a function by an increment

In either the Manzyuk or the `*fix-three?*` implementation, if you adopt the `scmutils` semantics for `(* <scalar> <function>)`, then multiplying an increment by the functional return value of `((D f) x)` does indeed produce an 'increment in the result'.

You're effectively scaling any eventual output by the incremental input. This makes sense because the eventual non-function outputs represent sensitivities of that output to the original `x`.

4.4 Return a function and its tangent?

If you want to implement a function that returns both the function value *and* the derivative-taking function, don't return the entire pair until the end, when `tg` produces a function that returns a non-function.

Alternatively, you could return a pair of

- non-perturbed function
- curried derivative-taking function

at each step, always offering the user a chance to bail out of the tangent graph. This is what the true `pushforward` operator would do.

4.5 Equality, Identity, Referential Transparency

The derivative with `*fix-three?*` is not a referentially transparent operator because it's behavior is tied up with how you build your program graph. Say you let-bind some `x` to `a1` and `a2`, then call `((D f) a1)` and `((D f) a2)`. Each call will give you the sensitivity to its binding without knowing that `a1` equals `a2`.

Should `D` be referentially transparent? Arguably not in the "Calculus on Graphs" interpretation described in 2.6.1, since you are attempting to calculate the sensitivity of a graph to one of its function-valued nodes. Making the graph larger by currying, for example, should change the sensitivities of the outputs.

4.6 What about (D ((D f) x))?

What if we call `D` on a function returned by `((D f) x)`?

Here is a nice test case. If `((D f) x)` returns a function, calling `D` on *that* produces a function that will eventually produce a mixed partial:

```
(let* ((f (lambda (x)
            (lambda (y)
              (lambda (z)
                (* x x y y z z)))))
       (df  (D f))
       (dfx (D (df 'x))))
  (simplify
   ((dfx 'y) 'z)))
;;=> (* 4 x y (expt z 2))
```

5 References

Obviously there are many more references if we put this anywhere public! For now, consider this section a placeholder.

- Manzyuk et al. 2019
- notes, description in `implementation_updated.ss`