

Comparing Current Approaches to Automatic Differentiation in Programming Languages

Quan LONG

August, 2022

Abstract

The extended work of final research internship of MPRI M2 program, under supervising of **Dr.Damiano MAZZA**, *directeur de recherche, CNRS (INS2I)* in **Laboratoire d'Informatique de Paris Nord** of **Université Sorbonne Paris Nord**.

Differentiation is an essential computation in Machine learning and Deep learning, as the Dataset and parameter size are exploding in actual projects, they could easily get to trillions as we speak, with the structures getting more and more complicated, doing **Automatic Differentiation** (AD for short) linearly and without exploding in the calculation of sub-expressions (hence no duplication for the ones already calculated) is currently an active topic. Mathematicians, logicians, developers, etc are modeling this problem from different perspectives and giving different approaches.

Greatly distinct from each other they might seem to be, some of them are actually quite similar despite of the methodology context, while some of them are uniquely designed and being quite different from others. In this internship, we compare current approaches of AD methods in programming language based on Damiano's paper [BMP20] and its reformulation we developed during the comparison.

Contents

1	Overture : Introduction	1
1.1	Basic notions of differentiation	2
1.1.1	Derivative	2
1.1.2	Rules for Differentiation	2
1.2	Category theory Preliminaries	3
2	Prelude : Automatic Differentiation	3
2.1	Forward-mode : Dual Number method	3
2.2	Reverse-mode : Backpropagation algorithm	4
2.3	Symbolic Backpropagation	5
3	Allegro : Existing Approaches	5
3.1	Simply Typed Lambda-Calculus with Linear Negation	5
3.2	Categorical approach	6
3.3	Approaches more close to implementation	7
4	Intermezzo : Graphical language setup	7
4.1	Setup in String diagrams	7
4.2	Setup in Proof nets	8
4.3	Graphical standard AD	8
5	Adagio : Comparing BMP20 with CHAD	9
5.1	Construction of BMP20 Reverse-mode transformation $\overleftarrow{\mathbf{D}}_a(t)$	9
5.2	Construction of CHAD Reverse-mode transformation $\overleftarrow{\mathbf{D}}_{\Gamma}(t)$	10
5.3	Comparing Graphical representations of BMP20 and CHAD	12
5.3.1	$op(t_1, \dots, t_k)$	12

5.3.2	let $x = t$ in s	13
5.3.3	$\lambda x.t$	14
5.3.4	$t s$	14
5.4	Comparison with an example	15
6	Adagio : Comparing reformulated BMP20 with Kra+22	16
6.1	Monadic approach with Kra+22	17
6.2	Reformulation of linear factoring for BMP20	18
6.2.1	New backpropagator	18
6.2.2	Typing	18
6.2.3	Typing rules	18
6.2.4	Evaluation rules	19
6.2.5	Verifying new transformation by graphical language	19
7	Presto : Conclusion	20
A	Conventions	i
A.1	Higher-order	i
A.1.1	Higher-order quantification (polymorphism)	i
A.1.2	Higher-order types	ii
A.1.3	Higher-order derivatives	ii
B	Intuition for the proof of the fact mentioned in BMP20	ii
C	The Typing rules of original BMP20 approach	ii
D	The Monadic translation and Reverse-mode wrapper of Kra+22	iii

1 Overture : Introduction

There's a fable back in the epoch of the Three Kingdoms: One day, King of Wu offered King of Wei, Cao Cao, an elephant, such a giant creature, Cao Cao thinks, and he wants to know exactly how much weight it has. But they didn't have steelyard big enough for an elephant, a chancellor offers an idea to cut it in pieces and weight them, which got denied: obviously a whole elephant is still required after weighting. The child Prince Cao Chong comes up with an idea, to put the elephant on a boat, then we only need to weight the things that makes the same waterline as the elephant does.

That gives us an insight of the basic idea of AD, especially backward AD – we don't want to duplicate and cut the elephant into pieces as it will be too difficult to make the elephant whole again, instead, we use dual number method to calculate and store some information of it, thus we'll be able to get back with much less efforts (linear).

In *Deep learning*, how to efficiently training a neural network involves efficiently computing gradients (more generally, Jacobians), by the time of this report, AD techniques are routinely used in the industry through deep learning frameworks such as TensorFlow [Aba+16], PyTorch [Pas+17] and most recently, JAX.¹

We won't talk too much about deep learning, but its optimization regarding the efficiency in terms of the *gradient descent algorithm* and *Monte-Carlo integration algorithms* rely crucially on the calculation of derivatives, and is where we are interested when it comes to AD, whose key idea is to compute the gradient of a *computational graph* by accumulating in a suitable way the partial derivatives of the basic functions composing the graph. The details and examples of numerical and symbolical (computational graph) of AD will be presented later in Prelude : Automatic Differentiation.

Among the different approaches of modeling AD, in a purely functional point of view, two problems are mostly concerned by us: does this method give a linear solution? This is the basic requirement the a reverse-mode, and furthermore, also the core point of this internship, does this method have sub-expressions sharing so that we don't need to deal with duplicated nodes. Some problems which are not the main concern such as how does the approach handle higher-order, for definition of higher-order see Higher-order, as it's normally natural for functional methods to go from first-order to higher-order.

Why it's very necessary to find a way to understand their mechanisms and compare them?

And per [VS21], correctness proofs of reverse AD have taken a define-by-run approach or have relied on non-standard operational semantics, using forms of symbolic execution [BMP20] [AP20] [MO20] . Most work that treats reverse AD as a source-code transformation does so by making use of complex transformations which introduce mutable state and/or non-local control flow. *As a result, we are not sure whether and why such techniques are correct.*

To understand the differences and mechanisms of different methods, we first developed a formal graphical language, could be either *string diagram* or *proof net*, in Intermezzo : Graphical language setup, it is very handy in comparing [BMP20] with [VS21], as they come from lambda calculus, linear substitution and category theory contexts, and could be further applied to [BAJM08] etc.

But for some approaches more close to implementation, such as [Kra+22], using only graphical language will not be a good only option, as it would get very complicated once some complex functions of real world functional languages (Haskell in this case) gets involved. However, in this case we can easily analysis and evaluate its mechanisms from the functions structures and outcomes.

During the comparisons, not only did we succeed in understanding the mechanisms, eventually we developed a reformulation of [BMP20] to solve the problem of sharing in sub-expressions.

The main contribution of this work consists of two parts:

- Comparing Simply Typed Lambda-Calculus with Linear Negation method [BMP20], which is the baseline of comparison, to categorical method, more specifically Combinatory Homomorphic AD (CHAD)[VS21], using a formal graphical language setup, to find out how CHAD solved the problem of sharing in sub-expressions.
- Comparing [BMP20] to [Kra+22], with a reformulation of linear factoring for [BMP20] constructed during the comparison.

¹<https://jax.readthedocs.io/en/latest/notebooks/quickstart.html>

1.1 Basic notions of differentiation

1.1.1 Derivative

Definition 1.1. Derivative $f' x$ of a function $f :: \mathbb{R} \rightarrow \mathbb{R}$ at a point x in the domain of f is a number defined as:

$$f' x = \lim_{\epsilon \rightarrow \infty} \frac{f(x + \epsilon) - f x}{\epsilon}$$

Beyond functions of type $\mathbb{R} \rightarrow \mathbb{R}$?

- complex numbers $\mathbb{C} \rightarrow \mathbb{C}$: works, division is defined.
- $\mathbb{R} \rightarrow \mathbb{R}^n$: works, dividing a vector in \mathbb{R}^n by a scalar.
- $\mathbb{R}^m \rightarrow \mathbb{R}^n$ troublesome, dividing a vector $\epsilon :: \mathbb{R}^m$ is not defined. Partial derivatives with respect to m scalar components of domain \mathbb{R}^m on codomain \mathbb{R}^n , hence a *Jacobian* matrix $\mathbf{J}_{ij} = \partial f_i / \partial x_j$ for $i \in \{1, \dots, n\}, j \in \{1, \dots, m\}$.

Scalar chain rule is omitted, vector chain rule: "multiplying" two matrices \mathbf{A} and \mathbf{B} (Jacobians):

$$(\mathbf{A} \cdot \mathbf{B})_{ij} = \sum_{k=1}^m \mathbf{A}_{ik} \cdot \mathbf{B}_{kj}$$

1.1.2 Rules for Differentiation

[Ell18]

$$\mathcal{D} :: (a \rightarrow b) \rightarrow (a \rightarrow (a \rightarrow b))$$

It follows that differentiating twice is:

$$\mathcal{D}^2 = \mathcal{D} \circ \mathcal{D} :: (a \rightarrow b) \rightarrow (a \rightarrow (a \rightarrow a \rightarrow b))$$

Theorem 1.2 (compose/"chain" rule).

$$\mathcal{D}(g \circ f) a = \mathcal{D} g (f a) \circ \mathcal{D} f a$$

For $f :: a \rightarrow b$ and $g :: b \rightarrow c$, then $\mathcal{D} f a :: a \rightarrow b$, and $\mathcal{D} g (f a) :: b \rightarrow c$, so both side has type of $a \rightarrow c$

Strictly speaking, it's not a compositional recipe for differentiating sequential compositions, i.e., $\mathcal{D}(g \circ f)$ doesn't only need $\mathcal{D} g$ and $\mathcal{D} f$, it also needs f . To restore compositionality, using:

$$\begin{aligned} \mathcal{D}^+ :: (a \rightarrow b) &\rightarrow (a \rightarrow b \times (a \rightarrow b)) \\ \mathcal{D}^+ f a &= (f a, \mathcal{D} f a) \end{aligned}$$

Corollary 1.3. \mathcal{D}^+ is efficiently compositional with respect to \circ .

$$\mathcal{D}^+(g \circ f) a = \mathbf{let} \{ (b, f') = \mathcal{D}^+ f a; (c, g') = \mathcal{D}^+ g b \} \mathbf{in} (c, g' \circ f')$$

Parallel Composition:

Theorem 1.4 (cross rule).

$$\mathcal{D}(f \times g) (a, b) = \mathcal{D} f a \times \mathcal{D} g b$$

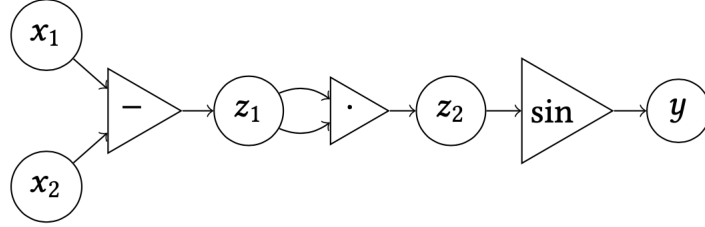
Definition 1.5 (linear). A function f is said to be linear when it distributes over (preserves the structure of) vector addition and scalar multiplication:

$$\begin{aligned} f(a + a') &= f a + f a' \\ f(s \cdot a) &= s \cdot f a \end{aligned}$$

Theorem 1.6 (linear rule). For all linear functions f , $\mathcal{D} f a = f$.

And therefore we have:

Corollary 1.7. For all linear functions f , $\mathcal{D}^+ f = \lambda a \rightarrow (f a, f)$.



let $z_1 = x_1 - x_2$ in let $z_2 = z_1 \cdot z_1$ in sin z_2

Figure 1: A computational graph with inputs x_1, x_2 and output y , and its corresponding term. Nodes are drawn as circles, hyperedges as triangles. The output y does not appear in the term: it corresponds to its root.

1.2 Category theory Preliminaries

We assume the readers are familiar with categories, functors, natural transformations, and their theory of (co)limits and adjunctions.

Definition 1.8. (Commutative Monoids) A monoid $|X|, 0_X, +_X$ consists of a set $|X|$ with an element $0_X \in |X|$ and a function $(+_X) : |X| \times |X| \rightarrow |X|$ such that $0_X +_X x = x = x +_X 0_X$ for any $x \in |X|$ and $x +_X (x' +_X x'') = (x +_X x') +_X x''$ for any $x, x', x'' \in |X|$. A monoid is called commutative if $x +_X x' = x' +_X x$ for all $x, x' \in |X|$. Given monoids X and Y , a function $f : |X| \rightarrow |Y|$ is called a homomorphism of monoids if $f(0_X) = 0_Y$ and $f(x +_X x') = f(x) +_Y f(x')$. [VS21] write **CMon** for the category of commutative monoids and their homomorphisms.

Grothendieck Constructions is within the preliminaries because CHAD method is based on it. But I will not waste a lot of space to describe it here since it's not an internship on category theory and it has very little connection to the main content. Anyone interested could check the definition here: <https://ncatlab.org/nlab/show/Grothendieck+construction>.

2 Prelude : Automatic Differentiation

Follow the basic notions of derivative in the introduction, now we can show that, given a program $x : r^n \vdash t : r^m$ that takes an n -dimensional array of real numbers as input and produces an m -dimensional array of reals as output, in another word, program t computes some mathematical function $\llbracket t \rrbracket : \mathbb{R}^n \rightarrow \mathbb{R}^m$ and we want to transform it to:

- $\vec{D}(t)$ that computes the derivative $\mathbf{D}\llbracket t \rrbracket : \mathbb{R}^n \rightarrow \underline{\mathbb{R}}^n \multimap \underline{\mathbb{R}}^m$, as the forward AD;
- $\overleftarrow{D}(t)$ that computes the transposed derivative $\mathbf{D}\llbracket t \rrbracket^t : \mathbb{R}^n \rightarrow \underline{\mathbb{R}}^m \multimap \underline{\mathbb{R}}^n$, as the reverse AD.

Here, we write $\underline{\mathbb{R}}^n$ for the space of (co)tangent vectors to \mathbb{R}^n ; we regard $\underline{\mathbb{R}}^n$ as a commutative monoid under elementwise addition. We write \multimap for a linear function type to emphasize that derivatives are linear in the sense of being monoid homomorphisms.[VS21]

Remark. Majority of the following content in this section are directly from [BMP20], part 2, **A CRASH COURSE IN AUTOMATIC DIFFERENTIATION**, to help the reader understand AD from scratch with example that we'll be using for the comparison. It is where I started to understand AD in the beginning of the internship with the help of Damiano as well.

2.1 Forward-mode : Dual Number method

Numerically, also in the implementations, we use the famous *Dual number method* to do forward AD. It is accomplished by augmenting the algebra of real numbers and obtaining a new arithmetic. An additional component is added to every number to represent the derivative of a function at the number, and all arithmetic operators are extended for the augmented algebra. The augmented algebra is the algebra of dual numbers.

Suppose that we are given a *computational graph* (in the sense of a hypergraph) G with input nodes x_1, \dots, x_n and one output node y , an example is as in Figure 1, and suppose that we want to compute its j -th partial derivative in $r = (r_1, \dots, r_n) \in \mathbb{R}^n$. It maintains a memory consisting of a set of assignments of the form

$x := (s, t)$, where x is a node of G and $s, t \in \mathbb{R}$ (known as **primal** and **tangent**), and proceeds as follows:

- we initialize the memory with $x_i := (r_i, 0)$ for all $0 \leq i \leq n, i \neq j$, and $x_j := (r_j, 1)$.
- At each step, we look for a node $z_1, \dots, z_k \rightarrow w$ such that $z_i = (s_i, t_i)$ is in memory for all $1 \leq i \leq k$ (there is at least one by acyclicity) and w is unknown, and we add to memory as a pair:

$$w := (f(\mathbf{s}), \Sigma_1^k \partial_i f(\mathbf{s}) \cdot t_i)$$

where $\mathbf{s} := s_1, \dots, s_m$.

This procedure terminates in a number of steps equal to $|G|$ and one may show, using the chain rule for derivatives (which we will recall in a moment), that at the end the memory will contain the assignment $y := ([G](r), \partial_j [G](r))$.

Example. For example, if G is the computational graph of Fig. 1 and we start with $x_1 := (5, 1), x_2 := (2, 0)$, we obtain $z_1 := (x_1 - x_2, 1 \cdot 1 - 1 \cdot 0) = (3, 1)$, then $z_2 := (z_1 \cdot z_1, z_1 \cdot 1 + z_1 \cdot 1) = (9, 6)$ and finally $y := (\sin(z_2), \cos(z_2) \cdot 6) = (0.412, -5.467)$, which is what we expect since $\partial_1 [G](x_1, x_2) = \cos((x_1 - x_2)^2) \cdot 2(x_1 - x_2)$.

Remark (remark on complexity and efficiency). Since the arity k of function symbols is bounded, the cost of computing one partial derivative is $O(|G|)$. Computing the gradient requires computing all n partial derivatives, giving a total cost of $O(n|G|)$, which is not very efficient since n may be huge (typically, it is the number of weights of a deep neural network, which may well be in the millions).

2.2 Reverse-mode : Backpropagation algorithm

Then we move to the reverse-mode, a more efficient way of computing gradients in the many inputs/one output case.

As usual, we are given a computational graph (seen as a hypergraph) G with input nodes x_1, \dots, x_n and output node y , as well as $r = r_1, \dots, r_n \in \mathbb{R}$, which is the point where we wish to compute the gradient. The backprop algorithm maintains a memory consisting of assignments of the form $x := (r, \alpha)$, where x is a node of G and $r, \alpha \in \mathbb{R}$ (the **primal** and the **cotangent**²), plus a boolean flag with values "marked/unmarked" for each hyperedge of G , and proceeds thus:

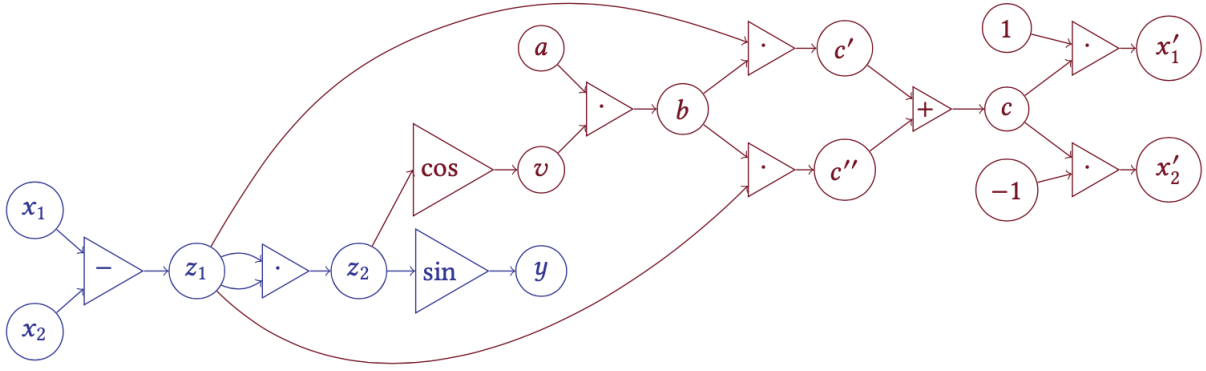
- **initialization:** the memory is initialized with $x_i := (r_i, 0)$ for all $1 \leq i \leq n$, and the forward phase starts;
- **forward phase:** at each step, a new assignment $z := (s, 0)$ is produced, with s being computed exactly as during forward evaluation, ignoring the second component of pairs in memory (i.e., s is the value of node z); once every node of G has a corresponding assignment in memory, the assignment $y := (t, 0)$ is updated to $y := (t, 1)$, every hyperedge is flagged as unmarked and the backward phase begins (we actually know that $t = [G](\mathbf{r})$, but this is unimportant);
- **backward phase:** at each step, we look for an unmarked hyperedge $z_1, \dots, z_k \rightarrow w$ such that all hyperedges having w among their sources are marked. If no such hyperedge exists, we terminate. Otherwise, assuming that the memory contains $w := (u, \alpha)$ and $z_i := (s_i, \beta_i)$ for all $1 \leq i \leq k$, we update the memory with the assignments $z_i := (s_i, \beta_i + \partial_i f(\mathbf{s}) \cdot \alpha)$ (where $\mathbf{s} := s_1, \dots, s_k$) for all $1 \leq i \leq k$ and flag the hyperedge as marked.

One may prove that, when the backprop algorithm terminates, the memory contains assignments of the form $x_i := (r_i, \partial_i [G](\mathbf{r}))$ for all $1 \leq i \leq n$, i.e., the value of each partial derivative of $[G]$ in \mathbf{r} is computed at the corresponding input node, and thus the gradient may be obtained by just collecting such values.

Example. Let $|G|$ be the computational graph of Fig. 1 and let $r_1 = 5, r_2 = 2$. The forward phase terminates with $x_1 := (5, 0), x_2 := (2, 0), z_1 := (3, 0), z_2 := (9, 0), y := (0.412, 1)$. From here, the backward phase updates $z_2 := (9, \cos(z_2) \cdot 1) = (9, -0.911)$, then $z_1 := (3, z_1 \cdot -0.911 + z_1 \cdot -0.911) = (3, -5.467)$ and finally $x_1 := (5, 1 \cdot -5.467) = (5, -5.467)$ and $x_2 := (2, -1 \cdot -5.467) = (2, 5.467)$, as expected since $\partial_2 [G] = -\partial_1 [G]$.

Remark (remark on complexity and efficiency). Compared to forward mode AD, the backprop algorithm may seem a bit contrived (it is certainly harder to understand why it works) but the gain in terms of complexity is considerable: the forward phase is just a forward evaluation; and, by construction, the backward phase scans each hyperedge exactly once performing each time a constant number of operations. So both phases are linear in $|G|$, giving a total cost of $O(|G|)$, like forward mode. Except that, unlike forward mode, a single evaluation now already gives us the whole gradient, independently of the number of inputs!

²The original paper uses "adjoint" but "cotangent" is the more popular way in the day of this report.



let $z_1 = x_1 - x_2$ in let $z_2 = z_1 \cdot z_1$ in let $b = (\cos z_2) \cdot a$ in let $c = z_1 \cdot b + z_1 \cdot b$ in $(\sin z_2, (1 \cdot c, -1 \cdot c))$

Figure 2: The computational graph of the $\mathbf{bp}_{x_1, x_2, a}(G)$ where G is in Fig. 1, and its corresponding term

2.3 Symbolic Backpropagation

The symbolic methodology we saw for forward mode AD may be applied to the reverse mode too: given a computational graph G with n inputs and one output, one may produce a computational graph $\mathbf{bp}(G)$ with $n + 1$ inputs and $1 + n$ outputs such that, for all $r = r_1, \dots, r_n \in \mathbb{R}$, the forward evaluation of $\mathbf{bp}(G)(\mathbf{r}, \mathbf{1})$ has output $(\llbracket G \rrbracket(\mathbf{r}), \nabla \llbracket G \rrbracket(\mathbf{r}))$. Moreover, $|\mathbf{bp}(G)| = O(|G|)$

Let us look at an example, shown in Fig. 2. First of all, observe that $\mathbf{bp}_{x_1, x_2, a}(G)$ contains a copy of G , marked in blue. This corresponds to the forward phase. The nodes marked in red correspond to the backward phase. Then we can check that the forward evaluation of $\mathbf{bp}_{x_1, x_2, a}(G)$ with $x_1 := 5, x_2 := 2$ and $a := 1$ matches exactly the steps of the backprop algorithm as exemplified in Reverse-mode : Backpropagation algorithm, with node b (resp. c, x'_1, x'_2) corresponding to the second component of the value of node z_2 (resp. z_1, x'_1, x'_2). (The nodes v, c' and c'' are just intermediate steps in the computation of b and c which are implicit in the numerical description of the algorithm and are hidden in syntactic sugar).

Different formal definitions of the reverse transformations will be shown and compared in following sections, and an extended version of [BMP20] solving the sharing problem in sub-expressions will be given in Reformulation of linear factoring for BMP20.

3 Allegro : Existing Approaches

There's another fable back in days, another elephant, was put in a room of completely darkness. Several people are asked to describe it, one touches the nose and says the elephant's like a giant snake, one touches the leg and says the elephant's like a tree.

From many different approaches we can achieve a linearized AD, but however it can be very difficult for others to understand why and how they work, how much different between them and how can they transfer to each other.

The earliest work performing AD in a functional setting is **Reverse-mode AD in a functional framework: Lambda the ultimate backpropagator** [BAJM08] in 2008, they want to define a programming language with the ability to perform AD on its own programs, by using a combinator $\overleftarrow{\mathcal{J}}$ computing the Jacobian of its argument, and whose execution implements reverse mode AD.

Why are we focusing on functional ?

ML, DL projects usually run on parallel devices, which purely functional is more adaptable.

3.1 Simply Typed Lambda-Calculus with Linear Negation

This approach was brought by Alois Brunel, Damiano Mazza, Michele Pagani in **Backpropagation in the Simply Typed Lambda-Calculus with Linear Negation** [BMP20], 2020, and is the baseline of the comparison of this internship.

The key to modular and efficient differentiable programming is making symbolic backprop (the reverse-mode transformation) compositional. And it achieved that by considering a programming language with a notion of

linear negation, and provides a method not only on computational graphs, or first-order linear function, but also possible higher order.

Its archetypal contravariant operation is negation:

Definition 3.1 (negation). For a (real) vector space A , negation corresponds to the dual space $A \multimap \mathbb{R}$, which may be generalized to $A^{\perp E} := A \multimap \mathbb{R}$ for an arbitrary space E , although in fact we will always take $E = \mathbb{R}^d$ for some $d \in \mathbb{N}$. We can keep E implicit if the context is not very concerned.

For forward-mode transformation, a canonical way of transforming a differentiable function $f : \mathbb{R} \rightarrow \mathbb{R}$ with derivative f' into a function $\mathbf{D}_r f : \mathbb{R} \times \mathbb{R}^\perp \rightarrow \mathbb{R} \times \mathbb{R}^\perp$ from which the derivative of f may be extracted. Namely, let, for all $x \in \mathbb{R}$ and $x^* \in \mathbb{R}^\perp$,

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

where they use λ -notation with the standard meaning.

And \mathbf{D}_r is compositional: for all $x \in \mathbb{R}$ and $x^* \in \mathbb{R} \multimap \mathbb{R}$, we have

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

This observation may be generalized to maps $f : \mathbb{R}^n \rightarrow \mathbb{R}$: for all $x \in \mathbb{R}^n$ and $\mathbf{x}^* = x_1^*, \dots, x_n^* \in \mathbb{R}^\perp$, hence the reverse-mode transformation

$$\overleftarrow{\mathbf{D}}(f)(x; \mathbf{x}^*) := (f(x), \lambda a^{\mathbf{R}. \Sigma_{i=1}^k x_i^*} (\partial_i f(x) \cdot a))$$

where the x_i^* here are the *backpropagators*.

Obviously $\overleftarrow{\mathbf{D}}(f) : (\mathbb{R} \times \mathbb{R}^\perp)^n \rightarrow \mathbb{R} \times \mathbb{R}^\perp$, and if we take $E = \mathbb{R}^n$, we have

$$(\pi_2 \overleftarrow{\mathbf{D}}(f)(\mathbf{x}; \iota_1, \dots, \iota_n))1 = \nabla f(\mathbf{x})$$

, where, for all $1 \leq i \leq n$, $\iota_i : \mathbb{R} \rightarrow \mathbb{R}^n$ is the injection into the i -th component, i.e., $\iota_i(x) = (0, \dots, x, \dots, 0)$ with zeros everywhere except at position i . Moreover, $\overleftarrow{\mathbf{D}}$ is compositional. For the full construction see 5.1.

- To be proved³: The fact that $\overleftarrow{\mathbf{D}}$ is a cartesian closed 2-functor or, better, a morphism of cartesian 2-multicategories, obtained by freely lifting to λ -terms a morphism defined on computational graphs.

3.2 Categorical approach

A notable work by Elliott [Ell18] gives a categorical, purely functional, define-then-run version of reverse AD, but however the sequential and parallel composition of differentiable functions rely (respectively) on sequential and parallel composition of linear maps, and likewise for each other operation. And their techniques are limited to first-order programs over tuples of real numbers.

[Vyt+19] then proposes two possible extensions of Elliott's functional AD to accommodate higher-order functions. However, it does not address whether or why these extensions would be correct or establish a more general methodology for applying AD to languages with expressive features.

Matthijs Vákár and Tom Smeding introduces **Combinatory Homomorphic Automatic Differentiation** (CHAD)[VS21] and its proof of correctness. CHAD is based on the observation that Elliott's work in [Ell18] has a unique structure preserving extension that lets them perform AD on various expressive programming language features. Which is **Grothendieck Constructions on Strictly Indexed Categories**. For more details of how they elegantly pair primals with (co)tangents, categorically, see section 6 in [VS21](not included in this report).

Remark. This paper also describes Abstract Denotational Semantics (by the properties of **Syn**), Concrete Denotational Semantics (in **CMon**) and finally Operational Semantics, which we won't discuss here.

Their comments on symbolic method:

1. Such approaches firstly have the severe downside that they can suffer from interpretation overhead.
2. Secondly, the differentiated code cannot benefit as well from existing optimizing compiler architectures. As such, these AD libraries need to be implemented using carefully, manually optimized code, that, for example, does not contain any common sub-expressions. This implementation process is precarious and labour intensive.

³I was trying to prove it but later with the reformulation it's no longer necessary, intuitions for the proof see Appendix B

3. Furthermore, some whole-program optimizations that a compiler would detect go entirely unused in such systems.

Their transformation $\vec{D}(t)_2$ and $\overleftarrow{D}(t)_2$ on a program t is compositional, generate purely functional code and the code size grow linearly in the size of t .

Its correctness and solving the problem of sharing sub-expressions in a purely categorical way make us interested and is one of the main comparison we did, as to be found in Adagio : Comparing BMP20 with CHAD.

3.3 Approaches more close to implementation

There exists many approaches in the implementation level, a good example I could think of is **Automatic Differentiation via Algebraic Effects and Handlers**, by Jesse Sigal, who gave his talk during the workshop[Sig20]. This implementation approach uses functional programming languages, although primarily Haskell, he mentioned that OCaml has a good handling on algebra effects and has potentials.

However in this internship we don't focus on them since it's very difficult to check their properties only by implementation. While some approaches using functional language structures to help achieving the properties we desired.

So we studied **Kra+22: Provably Correct, Asymptotically Efficient, Higher-Order Reverse-Mode Automatic Differentiation**[Kra+22] as the second half of the internship and also designed a reformulation of linear factoring for [BMP20].

Its second version of reverse-mode transformation is similar to [BMP20], and it uses a *Wrapper* structure around the transformations with evaluation rules to deal with the data flow.

To achieve the efficient property (their way of saying no duplication calculation of sub-expressions), they use Moggi's call-by-value monadic translation on the wrapper, and add supporting functions for the monadic translation to eliminate the duplication.

4 Intermezzo : Graphical language setup

I first tried to compare them in a pure numerical way, which is the execution of the example program in Figure 1 by both of the reverse-mode transformations. But as you will see next, the constructions of the transformations are very complicated to process in a such method, especially for CHAD, there are many categorical structures that are difficult and confusing to be presented.

After being stuck for almost a month, Damiano gave me a graphical point of view and it's surprisingly fit to our task, as both methods are naturally adapted to formal graphical languages such as proof nets and string diagram – By tracing the backpropagators and analysing the structures of the diagrams, comparing is far more easier!

Using a formal visualized language such as string diagram and proof net to compare them is original although there exist a paper [AP+21] using string diagram to do AD base on the method of [BAJM08].

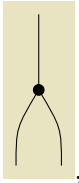
But first we need to setup a formal graphical language, to make sure they are not some subjective drafts.

4.1 Setup in String diagrams

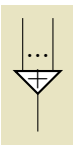
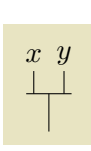
We will use the graphical syntax developed in [AP+21]. The full definition of string diagrams is omitted here, for the comparison we only need part of them.

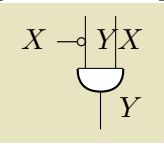
For more about the functorial boxes we're going to use to represent lambda abstraction and other operations see [Mel06].

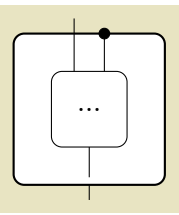
We represent type of a node in a circle, for example a node with type ι_k :  weakening as: ,

contraction as: ⁴, $\mathbf{R}^n \rightarrow \mathbf{R}$ arithmetic operations for example "plus" as a triangle that takes multiple

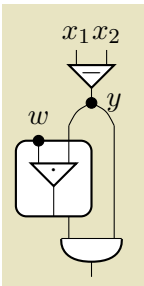
⁴The dot can be omitted.

inputs and gives one output:  , the tensor product of $x \otimes y$:  , higher-order application as a lower

semi-circle:  (this is also the $x \bullet y$ operation defined in [VS21]), and finally the lambda abstraction:

 , the variable of the lambda is the dot on the top right, it takes other symbols of the lambda term from the wire on the top.

Example. Let's start with an example to show how it works with **let** $y = x_1 - x_2$ **in** $(\lambda w. wy)y$:

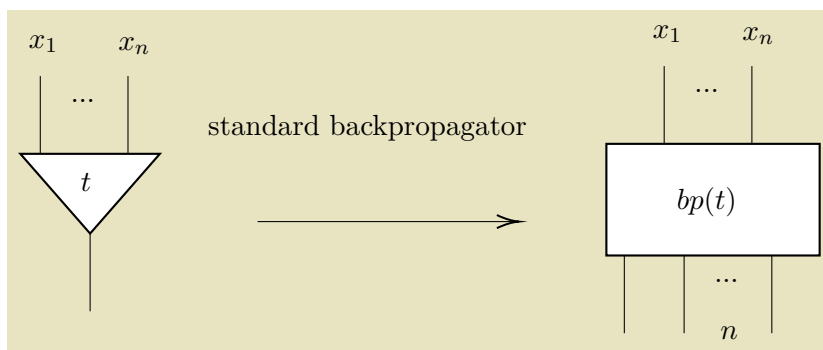
 y splits to two, one goes inside and form the lambda, while another applies to the lambda.

4.2 Setup in Proof nets

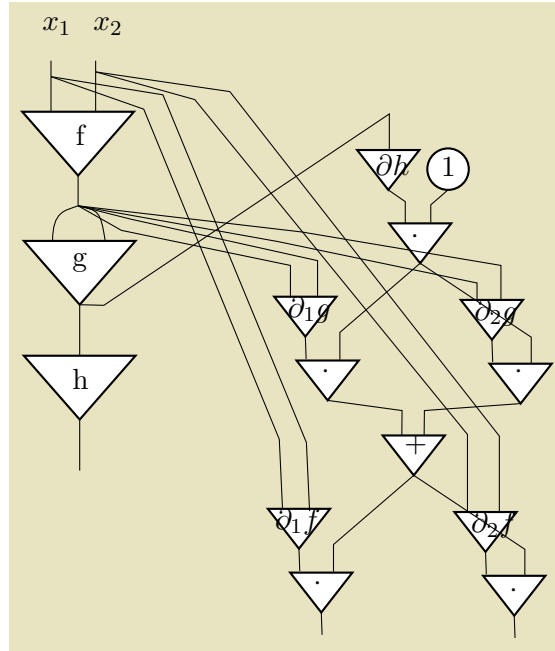
The graphical language could be easily adapted to Proof nets instead of String diagrams, for more details see [Gue01] and [Acc18].

4.3 Graphical standard AD

From the definition of AD of the last section, for a program t we can have:



Example. With execution on the example given in Figure 1:



5 Adagio : Comparing BMP20 with CHAD

As we have seen in section 3.1, although the method of lambda calculus with linear negation is compositional and elegantly simple, it doesn't naturally deal with duplication of sub-expressions. For this reason, efficiency is achieved only by considering a specific evaluation strategy. By contrast, CHAD has the ability to pass the duplicated sub-expression only once. We wanted to understand why it worked and how can we adapt similar method to BMP20.

We only compare the reverse-mode transformations to track with how each methods deal with cotangents, since the forward-mode is simply applying chain rule to each basic operations in a forward primal trace.

In this section, we'll give the detailed constructions on both methods, transfer them into the graphical language and explore their mechanisms with the right example.

5.1 Construction of BMP20 Reverse-mode transformation $\overleftarrow{D}_d(t)$

Definition 5.1. Terms and Types

Since the linear factoring rule^a is type-sensitive (it is unsound in general), it is convenient to adopt a Church-style presentation of our calculus, i.e., with explicit type annotations on variables. The set of types is generated by the following grammar:

$$\mathbf{A}, \mathbf{B}, \mathbf{C} ::= \mathbf{R} \mid \mathbf{A} \times \mathbf{B} \mid \mathbf{A} \rightarrow \mathbf{B} \mid \mathbf{R}^{\perp d} \quad (\text{simple types})$$

where \mathbf{R} is the ground type of real numbers.

^aThere will be a new linear factoring rule so we won't go deeply for the original one in this report

In this linear substitution calculus, the grammar of values and terms is given by mutual induction as follows:

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

$$t, u ::= v \mid ut \mid \langle t, u \rangle \mid t[x^{(!)A}, x^{(!)A}] := u \mid t[x^{(!)A} := u] \mid t + u \mid f(t_1, \dots, t_k) \quad (\text{terms})$$

The typing rules are not necessary for the comparing, however for those who are interested, see C.

Definition 5.2. Action of the transformation on types

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

Definition 5.3. Action of the transformation on terms

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

As we will also see in the following section of CHAD, to construct a transformation, we need to define: the single variable rule, lambda rule, application rule (tu), tuple rule, the **let** rule ($t[x^{!A} := u]$), empty rule, multiple operations rule ($f(t)$).

5.2 Construction of CHAD Reverse-mode transformation $\overleftarrow{\mathbf{D}}_{\Gamma}(t)$

They use **Linear λ -Calculus**, as an idealised target language, consider linear types (aka computation types), in addition to the Cartesian types (aka value types) τ, σ, ρ . They think of Cartesian types as denoting sets and linear types as denoting sets equipped with an algebraic structure.

Definition 5.4. Definition of $*$ and \bullet

$$\begin{aligned}
\llbracket \tau * \sigma \rrbracket &\stackrel{\text{def}}{=} \llbracket \tau \rrbracket \times \llbracket \sigma \rrbracket \\
\llbracket t \bullet s \rrbracket &\stackrel{\text{def}}{=} \underline{\Lambda}^{-1}(\llbracket t \rrbracket); \llbracket s \rrbracket \text{ (or } \llbracket s \rrbracket \circ \underline{\Lambda}^{-1}(\llbracket t \rrbracket))
\end{aligned}$$

Typing rules: Figure 3⁵

Evaluation rules: Figure 4

The suitable terms (e.g. linear operations) to represent the forward and reverse mode derivatives of the primitive operations $\text{op} \in \text{Op}_{n_1, \dots, n_k}^m$ are:

$$\begin{aligned}
x_1 : \mathbf{real}^{n_1}, \dots, x_k : \mathbf{real}^{n_k} ; v : \mathbf{real}^{n_1} * \dots * \mathbf{real}^{n_k} \\
\vdash \text{Dop}(x_1, \dots, x_k; v) : \mathbf{real}^m \\
x_1 : \mathbf{real}^{n_1}, \dots, x_k : \mathbf{real}^{n_k} ; v : \mathbf{real}^m \\
\vdash \text{Dop}^t(x_1, \dots, x_k; v) : \mathbf{real}^{n_1} * \dots * \mathbf{real}^{n_k}
\end{aligned}$$

Definition 5.5. Basic definitions

$$\begin{aligned}
\overleftarrow{\mathbf{D}}(\mathbf{real}^n)_1 &:= \mathbf{real}^n \\
\overleftarrow{\mathbf{D}}(\mathbf{real}^n)_2 &:= \mathbf{real}^n \\
\overleftarrow{\mathbf{D}}_{\Gamma}(\text{op}(t_1, \dots, t_k)) &:= \mathbf{let} \langle x_1, x'_1 \rangle = \overleftarrow{\mathbf{D}}_{\Gamma}(t_1) \mathbf{in} \dots \mathbf{let} \langle x_k, x'_k \rangle = \overleftarrow{\mathbf{D}}_{\Gamma}(t_k) \mathbf{in} \\
&\langle \text{op}(x_1, \dots, x_k), \\
&\lambda v. \mathbf{let} v^* = \text{Dop}^t(x_1, \dots, x_k; v) \mathbf{in} x'_1 \bullet (\mathbf{proj}_1 v^*) + \dots + \mathbf{in} x'_k \bullet (\mathbf{proj}_k v^*) \rangle
\end{aligned}$$

Implied reverse mode CHAD

⁵We found a typo and marked in red, the v here should be replaced by z .

$$\begin{array}{c}
\frac{}{\Gamma; \underline{\tau} \vdash \underline{v} : \underline{\tau}} \quad \frac{\Gamma \vdash t : \underline{\tau} \quad \Gamma, x : \underline{\tau}; \underline{v} : \underline{\sigma} \vdash s : \underline{\rho}}{\Gamma; \underline{v} : \underline{\sigma} \vdash \text{let } x = t \text{ in } s : \underline{\rho}} \quad \frac{\Gamma; \underline{v} : \underline{\tau} \vdash t : \underline{\sigma} \quad \Gamma; \underline{v} : \underline{\sigma} \vdash s : \underline{\rho}}{\Gamma; \underline{v} : \underline{\tau} \vdash \text{let } v = t \text{ in } s : \underline{\rho}} \\
\frac{\{\Gamma \vdash t_i : \text{real}^{n_i} \mid i = 1, \dots, k\} \quad \Gamma; \underline{v} : \underline{\tau} \vdash s : \text{LDom}(\text{lop}) \quad (\text{lop} \in \text{LOp}_{n_1, \dots, n_k; n'_1, \dots, n'_l}^{m_1, \dots, m_r})}{\Gamma; \underline{v} : \underline{\tau} \vdash \text{lop}(t_1, \dots, t_k; s) : \text{CDom}(\text{lop})} \\
\frac{}{\Gamma; \underline{v} : \underline{\tau} \vdash \langle \rangle : \underline{\mathbf{1}}} \quad \frac{\Gamma; \underline{v} : \underline{\tau} \vdash t : \underline{\sigma} \quad \Gamma; \underline{v} : \underline{\tau} \vdash s : \underline{\rho}}{\Gamma; \underline{v} : \underline{\tau} \vdash \langle t, s \rangle : \underline{\sigma * \rho}} \quad \frac{\Gamma; \underline{v} : \underline{\tau} \vdash t : \underline{\sigma * \rho}}{\Gamma; \underline{v} : \underline{\tau} \vdash \text{fst } t : \underline{\sigma}} \quad \frac{\Gamma; \underline{v} : \underline{\tau} \vdash t : \underline{\sigma * \rho}}{\Gamma; \underline{v} : \underline{\tau} \vdash \text{snd } t : \underline{\rho}} \\
\frac{\Gamma, y : \underline{\sigma}; \underline{v} : \underline{\tau} \vdash t : \underline{\rho}}{\Gamma; \underline{v} : \underline{\tau} \vdash \lambda y. t : \underline{\sigma} \rightarrow \underline{\rho}} \quad \frac{\Gamma; \underline{v} : \underline{\tau} \vdash t : \underline{\sigma} \rightarrow \underline{\rho} \quad \Gamma \vdash s : \underline{\sigma}}{\Gamma; \underline{v} : \underline{\tau} \vdash t s : \underline{\rho}} \quad \frac{\Gamma \vdash t : \underline{\sigma} \quad \Gamma; \underline{v} : \underline{\tau} \vdash s : \underline{\rho}}{\Gamma; \underline{v} : \underline{\tau} \vdash !t \otimes s : \underline{! \sigma \otimes \rho}} \\
\frac{\Gamma; \underline{v} : \underline{\tau} \vdash t : \underline{! \sigma \otimes \rho} \quad \Gamma, y : \underline{\sigma}; z : \underline{\rho} \vdash s : \underline{\rho'}}{\Gamma; \underline{v} : \underline{\tau} \vdash \text{case } t \text{ of } !y \otimes v \rightarrow s : \underline{\rho'}} \quad \frac{\Gamma; \underline{v} : \underline{\tau} \vdash t : \underline{\sigma}}{\Gamma \vdash \lambda v. t : \underline{\tau} \multimap \underline{\sigma}} \\
\frac{\Gamma \vdash t : \underline{\rho} \multimap \underline{\sigma} \quad \Gamma; \underline{v} : \underline{\tau} \vdash s : \underline{\rho}}{\Gamma; \underline{v} : \underline{\tau} \vdash t \bullet s : \underline{\sigma}} \quad \frac{}{\Gamma; \underline{v} : \underline{\tau} \vdash \underline{0} : \underline{\sigma}} \quad \frac{\Gamma; \underline{v} : \underline{\tau} \vdash t : \underline{\sigma} \quad \Gamma; \underline{v} : \underline{\tau} \vdash s : \underline{\sigma}}{\Gamma; \underline{v} : \underline{\tau} \vdash t + s : \underline{\sigma}}
\end{array}$$

Figure 3: Typing rules.

$$\begin{array}{l}
\text{let } v = t \text{ in } s = s[t/v] \\
\text{case } !t \otimes s \text{ of } !x \otimes v \rightarrow r = r[t/x, s/v] \quad t[s/x] \stackrel{\#y,z}{=} \text{case } s \text{ of } !y \otimes v \rightarrow t[!y \otimes v/x] \\
(\lambda v. t) \bullet s = t[s/v] \quad t = \lambda v. t \bullet v \\
t + \underline{0} = t \quad \underline{0} + t = t \quad (t + s) + r = t + (s + r) \quad t + s = s + t \\
(\Gamma; \underline{v} : \underline{\tau} \vdash t : \underline{\sigma}) \Rightarrow t[\underline{0}/v] = \underline{0} \quad (\Gamma; \underline{v} : \underline{\tau} \vdash t : \underline{\sigma}) \Rightarrow t[s+r/v] = t[s/v] + t[r/v]
\end{array}$$

Figure 4: Evaluation rules.

Definition 5.6. Types of (reverse mode) primals $\overleftarrow{\mathbf{D}}(\tau)_1$ and cotangents $\overleftarrow{\mathbf{D}}(\tau)_2$ associated with a type τ as :

$$\begin{array}{ll}
\overleftarrow{\mathbf{D}}(\mathbf{1})_1 & := \mathbf{1} & \overleftarrow{\mathbf{D}}(\mathbf{1})_2 & := \underline{\mathbf{1}} \\
\overleftarrow{\mathbf{D}}(\tau * \sigma)_1 & := \overleftarrow{\mathbf{D}}(\tau)_1 * \overleftarrow{\mathbf{D}}(\sigma)_1 & \overleftarrow{\mathbf{D}}(\tau * \sigma)_2 & := \overleftarrow{\mathbf{D}}(\tau)_2 * \overleftarrow{\mathbf{D}}(\sigma)_2 \\
\overleftarrow{\mathbf{D}}(\tau \rightarrow \sigma)_1 & := \overleftarrow{\mathbf{D}}(\tau)_1 \rightarrow (\overleftarrow{\mathbf{D}}(\sigma)_1 * (\overleftarrow{\mathbf{D}}(\sigma)_2 \multimap \overleftarrow{\mathbf{D}}(\tau)_2)) & \overleftarrow{\mathbf{D}}(\tau \rightarrow \sigma)_2 & := \overleftarrow{\mathbf{D}}(\tau)_1 \otimes \overleftarrow{\mathbf{D}}(\sigma)_2
\end{array}$$

Associate a non-trivial type of primals to function types as exponentials are not fibred in $\Sigma_{C_{syn}LSyn^{op}}$, see [VS21].

For programs t ,⁶

Definition 5.7. The reverse mode CHAD transformation $\overleftarrow{\mathbf{D}}_{\Gamma}(t)$ as follows:

⁶Different v in the same rule of CHAD definitions are not distinct from each other in the original paper, thus here we use v and v^* to separate them.

$$\begin{aligned}
\overleftarrow{\mathbf{D}}_{\overline{\Gamma}}(x) &:= \langle x, \underline{\lambda}v. \mathbf{coproj}_{\text{id}_x(x; \overline{\Gamma})}(v) \rangle \\
\overleftarrow{\mathbf{D}}_{\overline{\Gamma}}(\mathbf{let } x = t \text{ in } s) &:= \mathbf{let } \langle x, x' \rangle = \overleftarrow{\mathbf{D}}_{\overline{\Gamma}}(t) \text{ in } \mathbf{let } \langle y, y' \rangle = \overleftarrow{\mathbf{D}}_{\overline{\Gamma}, x}(s) \text{ in} \\
&\quad \langle y, \underline{\lambda}v. \mathbf{let } v^* = y' \bullet v \text{ in } \mathbf{fst } v^* + x' \bullet (\mathbf{snd } v^*) \rangle \\
\overleftarrow{\mathbf{D}}_{\overline{\Gamma}}(\langle \rangle) &:= \langle \langle \rangle, \underline{\lambda}v. \mathbf{0} \rangle \\
\overleftarrow{\mathbf{D}}_{\overline{\Gamma}}(\langle t, s \rangle) &:= \mathbf{let } \langle x, x' \rangle = \overleftarrow{\mathbf{D}}_{\overline{\Gamma}}(t) \text{ in } \mathbf{let } \langle y, y' \rangle = \overleftarrow{\mathbf{D}}_{\overline{\Gamma}}(s) \text{ in} \\
&\quad \langle \langle x, y \rangle, \underline{\lambda}v. x' \bullet (\mathbf{fst } v) + y' \bullet (\mathbf{snd } v) \rangle \\
\overleftarrow{\mathbf{D}}_{\overline{\Gamma}}(\mathbf{fst } t) &:= \mathbf{let } \langle x, x' \rangle = \overleftarrow{\mathbf{D}}_{\overline{\Gamma}}(t) \text{ in } \langle \mathbf{fst } x, \underline{\lambda}v. x' \bullet \langle v, \mathbf{0} \rangle \rangle \\
\overleftarrow{\mathbf{D}}_{\overline{\Gamma}}(\mathbf{snd } t) &:= \mathbf{let } \langle x, x' \rangle = \overleftarrow{\mathbf{D}}_{\overline{\Gamma}}(t) \text{ in } \langle \mathbf{snd } x, \underline{\lambda}v. x' \bullet \langle \mathbf{0}, v \rangle \rangle \\
\overleftarrow{\mathbf{D}}_{\overline{\Gamma}}(\lambda x. t) &:= \mathbf{let } y = \lambda x. \overleftarrow{\mathbf{D}}_{\overline{\Gamma}, x}(t) \text{ in} \\
&\quad \langle \lambda x. \mathbf{let } \langle z, z' \rangle = yx \text{ in} \\
&\quad \langle z, \underline{\lambda}v. \mathbf{snd } (z' \bullet v) \rangle, \underline{\lambda}v^*. \mathbf{case } v^* \text{ of } !x \otimes v \rightarrow \mathbf{fst } ((\mathbf{snd } (yx)) \bullet v) \rangle \\
\overleftarrow{\mathbf{D}}_{\overline{\Gamma}}(t s) &:= \mathbf{let } \langle x, x'_{ctx} \rangle = \overleftarrow{\mathbf{D}}_{\overline{\Gamma}}(t) \text{ in } \mathbf{let } \langle y, y' \rangle = \overleftarrow{\mathbf{D}}_{\overline{\Gamma}}(s) \text{ in } \mathbf{let } \langle z, x'_{arg} \rangle = xy \text{ in} \\
&\quad \langle z, \underline{\lambda}v. x'_{ctx} \bullet (!y \otimes v) + y' \bullet (x'_{arg} \bullet v) \rangle
\end{aligned}$$

The transformations for variables, tuples and projections implement the well-known multivariate calculus facts about (transposed) derivatives of differentiable functions into and out of products of spaces. The transformations for let-bindings add to the chain rules. The transformations for λ -abstractions split the derivative of a closure $\lambda x.t$ into the derivative z' with respect to x and the derivative $\mathbf{snd}(yx)$ with respect to the captured context variables; they store z' together with the primal computation z of $\lambda x.t$ in the primal associated with the closure and they store $\mathbf{snd}(yx)$ in the cotangent associated with the closure. Conversely, the transformations for evaluations extracts those two components of the (transposed) derivative x'_{ctx} (w.r.t. context variables) and x'_{arg} (w.r.t. function argument) from the cotangent and primal, respectively, and recombine them to correctly propagate cotangent contributions from both sources.

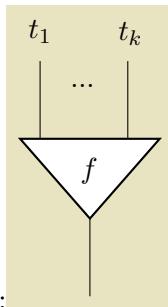
5.3 Comparing Graphical representations of BMP20 and CHAD

Of all the rules, the ones interesting are the construction of the multiple t_i rule, **let** rule, lambda rule and the application rule.

We will first compare them one by one then we can use a simple example to show their mechanisms.

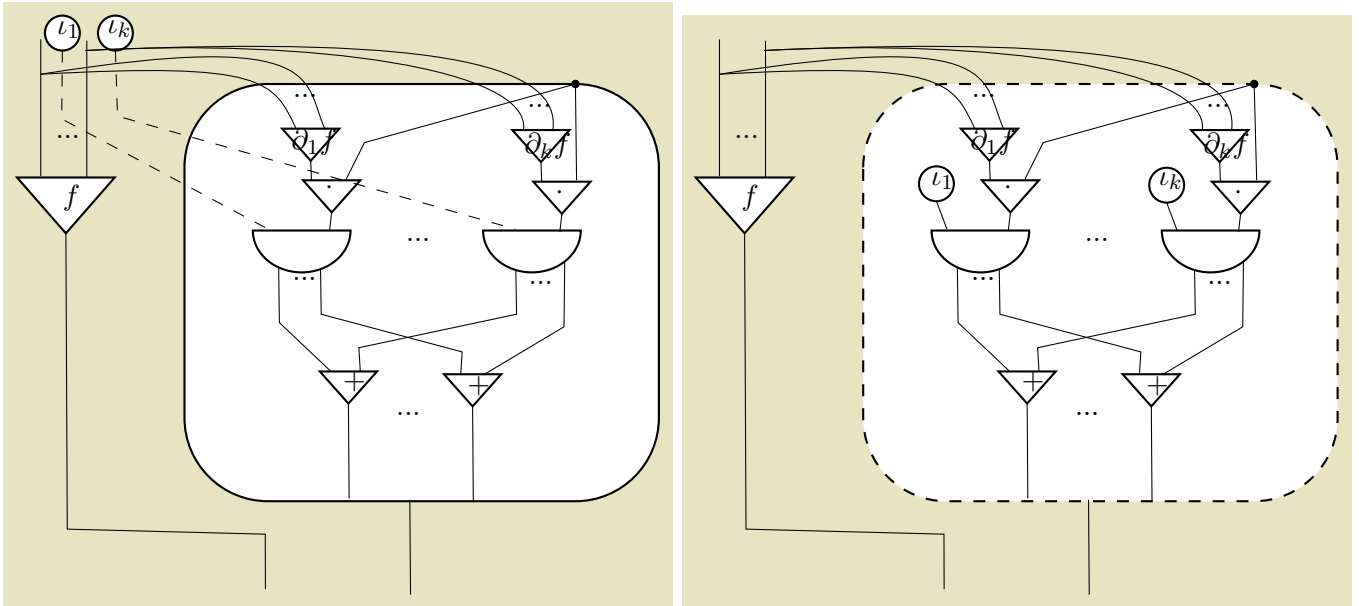
Remark. We use dash lines for the cotangents, and dash lambda is the linear lambda $\underline{\lambda}$ in CHAD.

5.3.1 $op(t_1, \dots, t_k)$



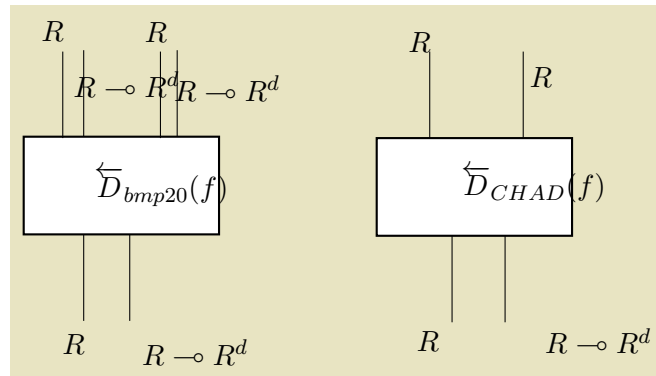
For $op(t_1, \dots, t_k)$:

Left is BMP20 and right is CHAD:



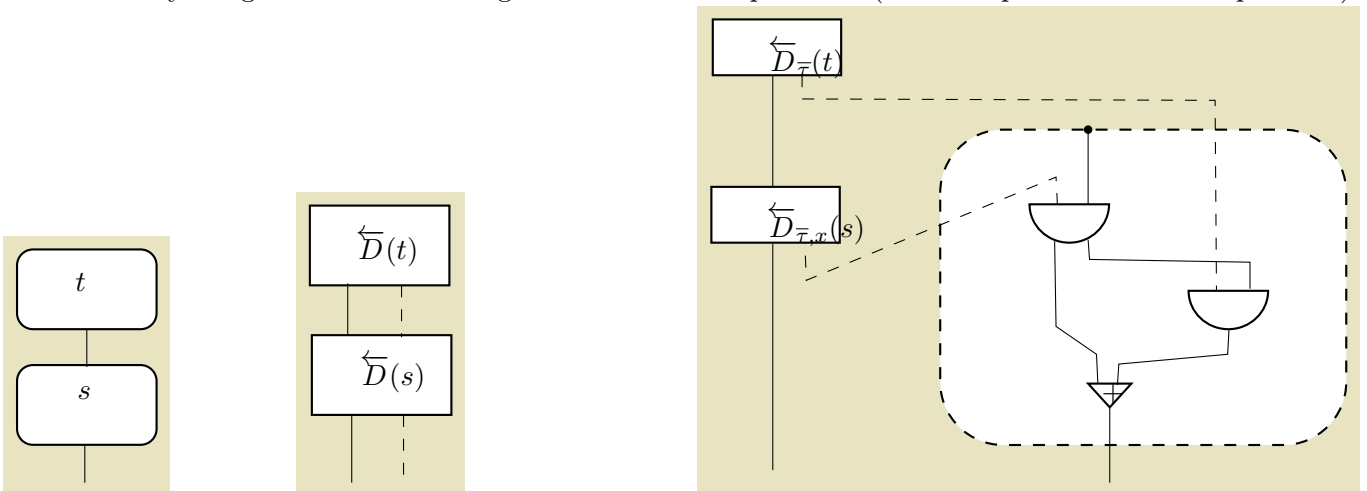
The l_i here has the type of $\mathbf{R} \multimap \mathbf{R}^d$, both methods use same type for backpropagators, but BMP20 extracts the l types directly from variables while CHAD keep this process inside the lambda.

We can also compare those diagrams in a meta level, to have a better look at their inputs and outputs, for example:

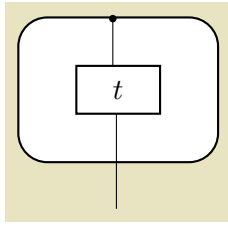


5.3.2 let $x = t$ in s

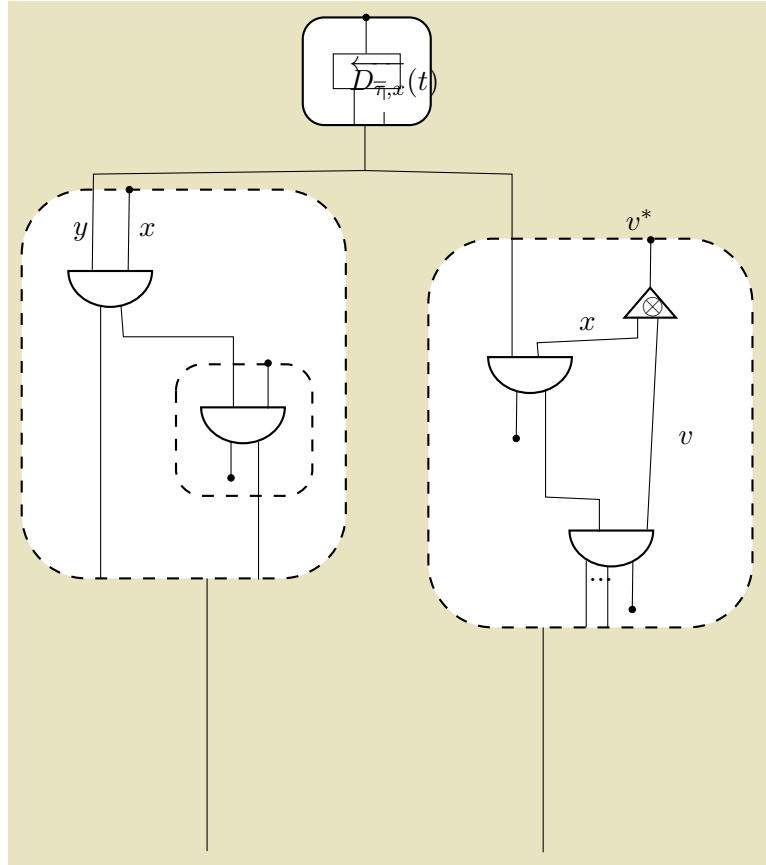
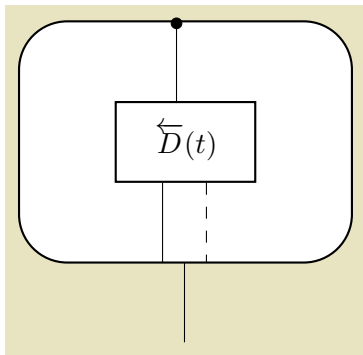
The **let** rule and application rule for BMP20 are very simple due to its functional structure, right is CHAD which is carefully designed to avoid sharing of common sub-expressions (we'll compare with an example later):



5.3.3 $\lambda x.t$

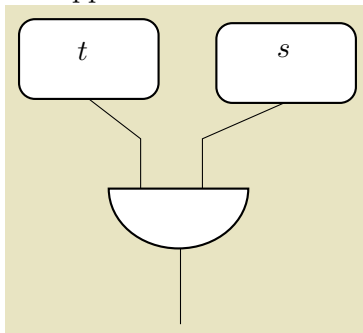


For lambda rule: ,left is BMP20 and right is CHAD:

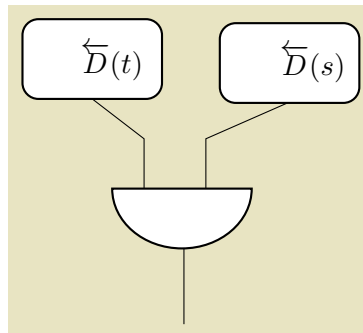


5.3.4 $t s$

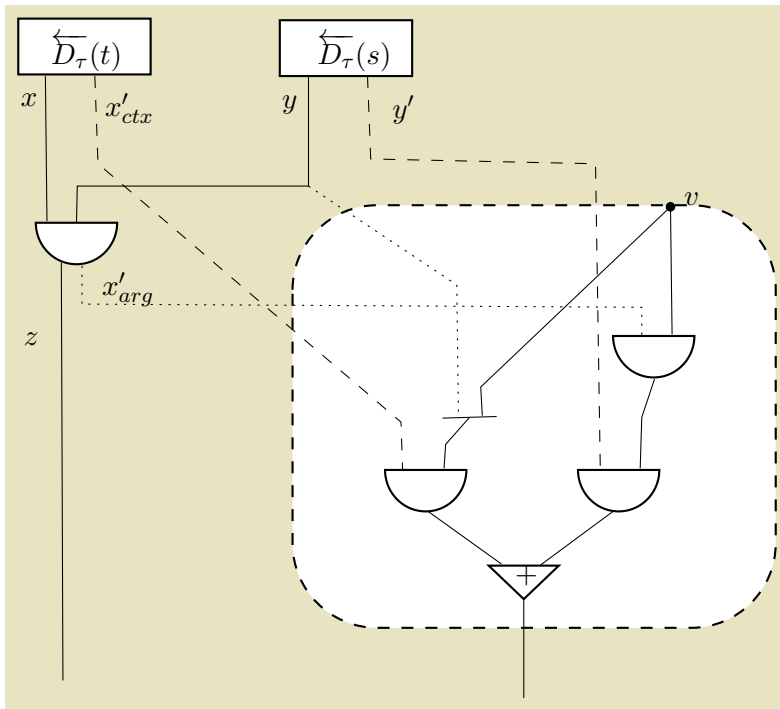
For application rule:



BMP20:



CHAD: In reality, we trace only the cotangents, the primal operations has very few impact, so we can ignore the inputs (which are dot lines) from primal to the lambda :



5.4 Comparison with an example

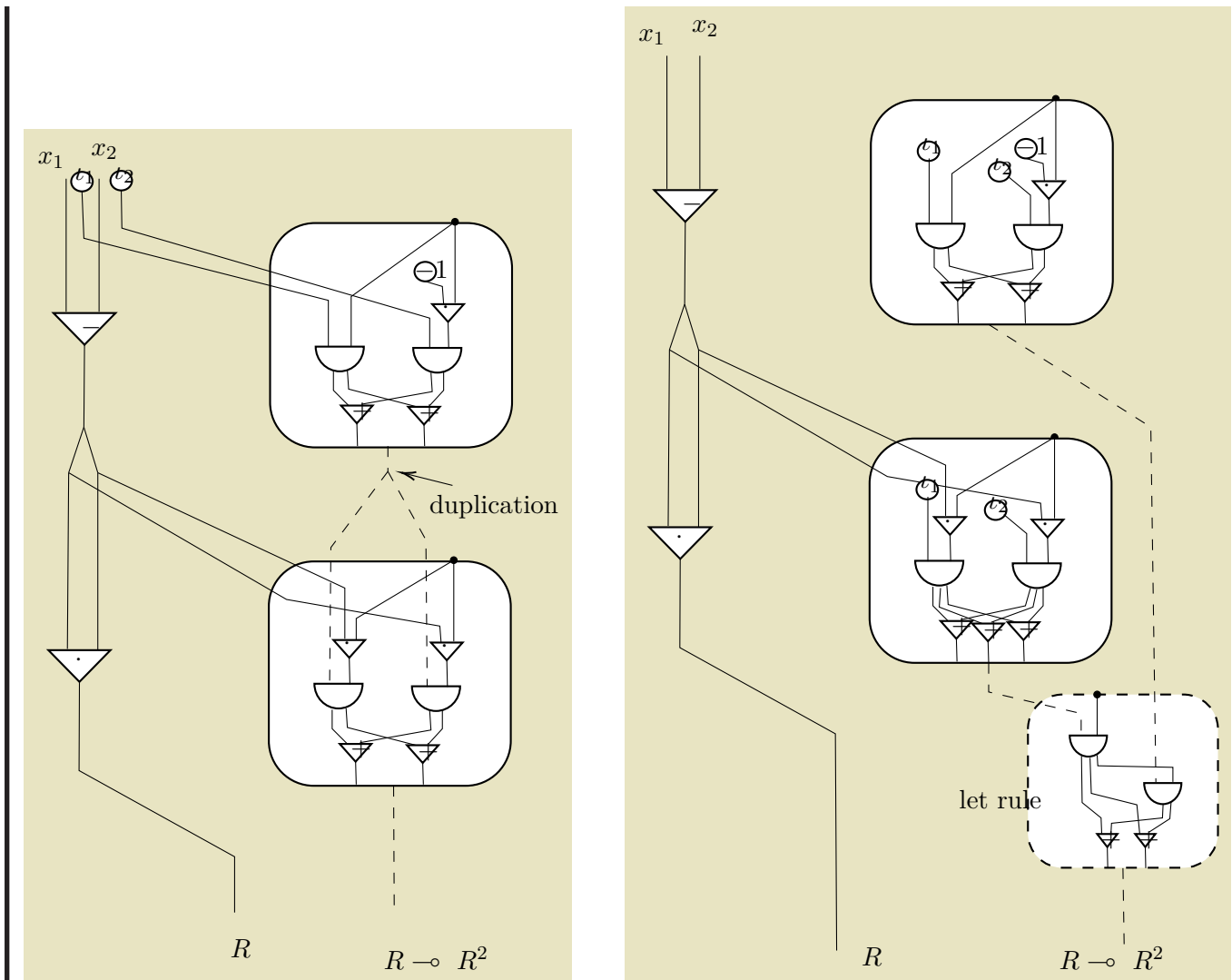
Now we have the full setup of both methods, it's time to study with an example similar to the problem that was raised in [BMP20] for duplication of sub-expressions :

$$\mathbf{let} \quad y = x_1 - x_2 \quad \mathbf{in} \quad z = y \cdot y \quad \mathbf{in} \quad z$$

The duplication of sub-expressions happens in the cotangent phase, hence we only need to check the dash strings.

Remark. For this property we only need to consider **let** rule and basic function rule for first order, Lambda and application rules are not considered.

Example. The left is BMP20 and right is CHAD:



This visualizes how they deal with the duplication of variable: BMP20 extracts the data type along with the variable, so duplication is unavoidable; while CHAD uses a special **let** construction accepting feeds from lambdas and process to a single result, there's no direct variable passing through the lambdas (which are different layers of functions), since all outputs of the lambdas and **let** are single with duplicated variables, it does not calculate twice !

6 Adagio : Comparing reformulated BMP20 with Kra+22

There are three versions of reverse-mode transformations in [Kra+22]: first one is a rough idea of backpropagation, second one is similar to [BMP20], also has sub-expression sharing problem, we won't go deep at the comparison in this level for two reasons:

1. They adopt several conveniences inspired by Haskell:
 - **do**-notion allows sequencing of monadic operations by using statements.
 - In the statement $x \leftarrow e$, when $e : M A$, then $x : A$.
 - The expression **pure** e has type $M A$ when $e : A$.
 - The type of the **do**-expression is the same as the type of the expression in the last statement.

we did try to use the graphical language to visualize them, but we found that to do it, we need to first model those functions in Haskell, and thus the diagram will be extremely complicated, in the other hand, due to its implementation oriented design, we could simply just look at the outcomes and its functions.

2. It's the solving of this problem, which is the monadic translation from second version to final version that interests us.

```

data Delta = Zero | Scale ℝ Delta | Add Delta Delta | Var Deltald | Let Deltald Delta Delta
type DeltaBinds = [(Deltald, Delta)] -- In reverse dependency order
type DeltaState = (Deltald, DeltaBinds)
type Deltald = ℕ
type DeltaMap = Map Deltald δℝ

eval : ℝ → Delta → DeltaMap → DeltaMap
eval x Zero          um = um
eval x (Scale y u)   um = eval (x × y) u um
eval x (Add u1 u2) um = eval x u2 (eval x u1 um)
eval x (Var uid)     um = addDelta uid x um
eval x (Let uid u1 u2) um = let um2 = eval x u2 um in
                               case lookup uid um2 of
                                   Nothing → um2
                                   Just x   → eval x u1 (delete uid um2)

-- API for the Map type
emptyMap : DeltaMap
lookup   : Deltald → DeltaMap → Maybe ℝ
lookupOrZero : Deltald → DeltaMap → ℝ
delete   : Deltald → DeltaMap → DeltaMap
addDelta : Deltald → ℝ → DeltaMap → DeltaMap
-- Adds to an existing entry, or create an entry if one does not exist

-- The monad M
type M a = DeltaState → (a, DeltaState)
runDelta : Deltald → M (ℝ, Delta) → (ℝ, Delta)
-- Runs the computation, and wraps the result in
-- the bindings produced by running the computation
runDelta delta_id m = (res, foldl wrap delta binds)
where
    ((res, delta), (_, binds)) = m (delta_id, [])
    wrap body (id, rhs)        = Let id rhs body

instance Monad M where
    return x = λs → (x, s)
    m ≻ n = λs → case m s of (r, s') → n r s'

deltaLet : Delta → M Deltald
deltaLet delta = λ(delta_id, bs) → (delta_id, (delta_id + 1, (delta_id, delta) : bs))

```

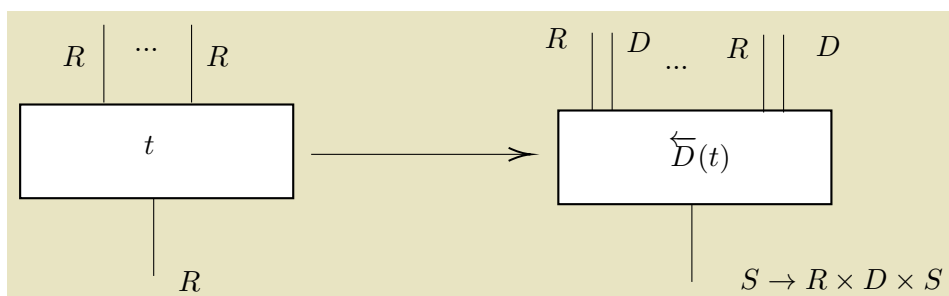
Figure 5: Supporting functions (rendered in Haskell-like syntax) for the monadic translation

6.1 Monadic approach with Kra+22

For us what's most interesting is their Monadic translation from the second version to the final version. Which solved the sub-expression problem.

Instead of using $\mathbf{R} \multimap \mathbf{R}$ like we've seen so far, it has a custom type **Delta** (as **D** in short) as the π_2 data in the dual number.

First let's look at the meta level using the graphical language, given a program t , we have:



where

$$\mathbf{S} := \mathbf{Nat} \times \mathit{List}(\mathbf{Nat} \times \mathbf{D})$$

The list structure is to assign id (\mathbf{Nat}) to each of the sub-expressions it passes.

For Monadic translation rules and Reverse-mode wrapper around the translation, see Appendix D. What we're interested is the supporting functions, for the monadic translation in Figure 5:

The last evaluation rule (the **let**) is how they deal with the duplication, when each sub-expression is processed, it checks the uid of this term in um_2 , if it hasn't been processed before, add it to um_2 , otherwise delete it.⁷

6.2 Reformulation of linear factoring for BMP20

In the beginning of this internship, when I first studied the sub-expression problem, I had the intuition that we can simply create a *storage* structure to check the sub-expressions for duplication. Kra+22 is using a similar structure in an implementation oriented approach to achieve it.

But can we do it in a more functional way?

Now we design a reformulation of linear factoring for BMP20 inspired by the *Wrapper* structure of Kra+22, adding evaluation rules and a modified backpropagator with a linear lambda l to store all the different encountered sub-expressions to \mathbf{C} (described in the last 2 evaluation rules). In implementation level, we can construct the backpropagator as a hash-table. More specifically, when a sub-expression is processed, check if the backpropagator x_l^* is already in \mathbf{C} , if not, process and add it to \mathbf{C} ; if it already exists, we don't pass it, because it should've already been passed.

6.2.1 New backpropagator

Definition 6.1. Introducing a new backpropagator x_l^* of type $\mathbf{R} \multimap \mathbf{D}$; d, e of type \mathbf{D} , which is the output type generated by the SOURCE type.

$$\mathbf{D} \cong \mathbf{R}^n$$

By applying t of type \mathbf{R} to the backpropagator, we can get x_l^*t of type \mathbf{D} . We can use $\mathbf{R}^{\perp \mathbf{D}}$ for short of $\mathbf{R} \multimap \mathbf{D}$. See negation.

6.2.2 Typing

Definition 6.2. linear lambda l ,

$$la.d$$

lambda which only process when a is linear.

This linear lambda makes sure the node passed is linear. Instead of values and terms style like in [BMP20], we have two calculi: SOURCE calculi with terms t, u and TARGET calculi dealing with terms d, e of type \mathbf{D} .

$$\begin{array}{lll} t, u & := & x \mid \langle t, u \rangle \mid \pi_1 t \mid \pi_2 t \mid \mathbf{let} \ x := u \ \mathbf{in} \ t \mid f(t_1, \dots, t_n) \mid la.d & \text{(SOURCE)} \\ d, e & := & 0 \mid d \oplus e \mid x_l^* t & \text{(TARGET)} \\ \mathbf{A}, \mathbf{B} & := & \mathbf{R} \mid \mathbf{R}^{\perp \mathbf{D}} \mid \mathbf{D} \mid \mathbf{A} \times \mathbf{B} & \text{(types)} \end{array}$$

Remark. $f(t_1, \dots, t_n)$: in short of 3 cases $r, t + u, t \cdot u$.

6.2.3 Typing rules

$$\begin{array}{c} \frac{}{\Gamma \vdash 0 : \mathbf{D}} \\ \frac{}{\Gamma; x : \mathbf{R} \vdash x : \mathbf{R}} \\ \frac{}{\Gamma; a : \mathbf{R} \vdash a : \mathbf{R}} \\ \frac{\Gamma; \Delta \vdash t : \mathbf{R} \quad \Gamma; \Delta \vdash u : \mathbf{R}}{\Gamma; \Delta \vdash t + u : \mathbf{R}} \end{array}$$

⁷In my opinion it's not necessary to to the delete since it's enough that the uid doesn't store in um_2 .

$$\frac{\Gamma; \Delta_1 \vdash t : \mathbf{R} \quad \Gamma; \Delta_2 \vdash u : \mathbf{R}}{\Gamma; \Delta_1; \Delta_2 \vdash t \cdot u : \mathbf{R}}$$

$$\frac{\Gamma; \Delta \vdash t : \mathbf{R}}{\Gamma; \Delta; x_i^* : \mathbf{R}^\perp \vdash x_i^* t : \mathbf{D}}$$

$$\frac{\Gamma; \Delta; a : \mathbf{R} \vdash d : \mathbf{D}}{\Gamma; \Delta \vdash la.d : \mathbf{R}^{\perp \mathbf{D}}}$$

$$\frac{\Gamma; \Delta \vdash d : \mathbf{D} \quad \Gamma; \Delta \vdash e : \mathbf{D}}{\Gamma; \Delta \vdash d \oplus e : \mathbf{D}}$$

$$\frac{\Gamma; \Delta_1 \vdash u : \mathbf{A} \quad \Gamma; \Delta_2; x : \mathbf{A} \vdash t : \mathbf{C}}{\Gamma; \Delta_1; \Delta_2 \vdash \mathbf{let} \ x := u \ \mathbf{in} \ t : \mathbf{C}}$$

Remark. context \mathbf{C} : it's a *one-hole context* same as the ones in the paper [BMP20], defined by the typing rules restricted to the terms having exactly one occurrence of a specific variable $\{\cdot\}$, called the *hole*. Given a context \mathbf{C} and a term t we denote by $\mathbf{C}\{t\}$ the substitution of the hole in \mathbf{C} by t allowing the possible capture of free variables of t .

6.2.4 Evaluation rules

8

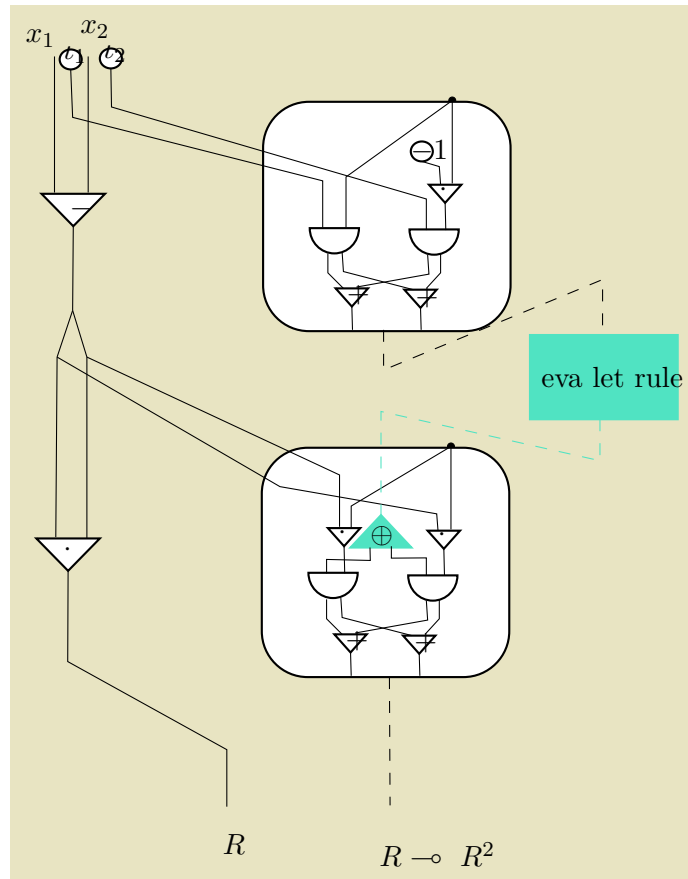
$$\begin{aligned} \pi_1 \langle t, u \rangle &\rightarrow t \\ \pi_2 \langle t, u \rangle &\rightarrow u \\ f(r_1, r_2, \dots, r_n) &\rightarrow S, \text{ if } S = f(r_1, r_2, \dots, r_n) \\ \mathbf{let} \ x := r \ \mathbf{in} \ t &\rightarrow t[r/x] \\ x_i^* t \oplus x_i^* u &\rightarrow x_i^* (t + u) \\ \mathbf{let} \ x_i^* := la.d \ \mathbf{in} \ \mathbf{C}\{x_i^* u\} &\rightarrow \mathbf{C}\{d[u/a]\} \text{ provided that } x_i^* \notin \mathbf{C} \end{aligned}$$

Remark. The construction of new backpropagation transformation $\overleftarrow{\mathbf{D}}_l$ with x_i^* is necessarily the same as the original one in 5.1.

6.2.5 Verifying new transformation by graphical language

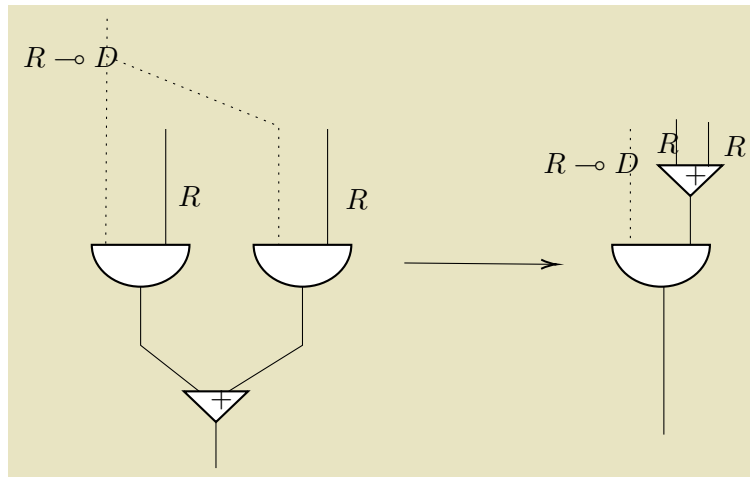
We can then verify the properties of the new transformation on the example of 5.4:

⁸ r in here means real number.



When we have processed the first lambda, instead of duplicate and process to the next lambda, we apply with the last evaluation rule, which checks the backpropagator with context \mathbf{C} , and process only once (since the duplicated one is already in \mathbf{C} after the first encounter), then apply with the \oplus evaluation rule in the next lambda with the backpropagator.

This is what happened in the second lambda by the \oplus evaluation rule doing linear factoring:



Remark. However, its formal validity proof is not yet studied, and should be done in the future.

7 Presto : Conclusion

Summary and future work are already mentioned in the first page as required.

I had a pretty rough start in the beginning of the internship, I just finished heavy exams and a very intense course semester, plus the topic itself requires a broad background and deep understandings in related disciplines which I did not possess. But when Damiano gives the idea of graphical language (I personally prefer visualized proofs as well), we progressed fast and thus lead to the results presented in this report. I learnt quite a lot from this internship and would love to continue with the studies in this topic.

References

- [Aba+16] Martín Abadi et al. “TensorFlow: A System for Large-Scale Machine Learning”. In: *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)* (2016). URL: <https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf>.
- [Acc18] Beniamino Accatooli. “Proof Nets and the Linear Substitution Calculus”. In: *15th International Colloquium on Theoretical Aspects of Computing (ICTAC 2018)* (2018). URL: <https://hal.archives-ouvertes.fr/hal-01967532/document>.
- [AP20] Martin Abadi, Gordon Plotkin. “A Simple Differentiable Programming Language”. In: *POPL, ACM* (2020).
- [AP+21] Mario Alvarez-Picallo et al. “Functorial String Diagrams for Reverse-Mode Automatic Differentiation”. In: (2021). URL: <https://arxiv.org/abs/2107.13433>.
- [BAJM08] Pearlmutter B. A., Siskind J. M. “Reverse-mode AD in a functional framework: Lambda the ultimate backpropagator”. In: *ACM Trans. Prog. Lang. Syst.* 30, 2, Article 7 (2008), p. 36. ISSN: 0019-9958. DOI: <https://doi.org/10.1145/1330017.1330018>.
- [BMP20] Aloïs Brunel, Damiano Mazza, Michele Pagani. “Backpropagation in the Simply Typed Lambda Calculus with Linear Negation”. In: *Proc. ACM Program. Lang.* 4, *POPL, Article 64* (2020). URL: <https://arxiv.org/abs/1909.13768>.
- [Ell18] Conal Elliott. “The Simple Essence of Automatic Differentiation”. In: *Proceedings of the ACM on Programming Languages* 2, *ICFP* (2018).
- [Gue01] Stefano Guerrini. “Proof nets and the λ -calculus”. In: (2001).
- [Kra+22] Faustyna Krawiec et al. “Provably Correct, Asymptotically Efficient, Higher-Order Reverse-Mode Automatic Differentiation”. In: *Proc. ACM Program. Lang.* 6, *POPL, Article 48* (2022).
- [Mel06] Paul-André Melliès. “Functorial Boxes in String Diagrams”. In: *Computer Science Logic, 20th International Workshop, CSL 2006, 15th Annual Conference of the EACSL, Szeged, Hungary, September 25-29, 2006, Proceedings (Lecture Notes in Computer Science)* (2006). URL: https://doi.org/10.1007/11874683_1.
- [MO20] Carol Mak, Luke Ong. “A Differential-form Pullback Programming Language for Higher-order Reverse-mode Automatic Differentiation”. In: (2020). URL: <https://arxiv.org/abs/2002.08241>.
- [Pas+17] Adam Paszke et al. “Automatic differentiation in PyTorch”. In: *31st Conference on Neural Information Processing Systems (NIPS 2017)* (2017). URL: <https://openreview.net/pdf?id=BJJsrnmcZ>.
- [Sig20] Jesse Sigal. “Automatic Differentiation via Algebraic Effects and Handlers”. In: (2020). URL: <https://lipn.univ-paris13.fr/~mazza/DiffProgWorkshop/Sigal.pdf>.
- [VS21] Matthijs Vákár, Tom Smeding. “CHAD: Combinatory Homomorphic Automatic Differentiation”. In: *ACM Trans. Program. Lang. Syst.* 1, 1, Article 1 (2021). URL: <https://arxiv.org/abs/2103.15776>.
- [Vyt+19] Dimitrios Vytiniotis et al. “The Differentiable Curry”. In: *NeurIPS 2019 Workshop Program Transformations* (2019).

A Conventions

A.1 Higher-order

A.1.1 Higher-order quantification (polymorphism)

(for second-order quantification, which is the one people are usually interested in when dealing with programming languages) The logical meaning. It means that quantifiers may range over properties, rather than just individuals. It is the "higher order" of System F.

A.1.2 Higher-order types

The programming languages meaning. It means that programs can take as argument other programs. In term of types, it means that, in the arrow type $A \rightarrow B$, the type A may itself be an arrow type. In fact, the order of a type is usually defined as follows: base types have order 0; and the order of $A \rightarrow B$ is one plus the maximum between the orders of A and B . So according to Damiano, "going from first order to higher order is for free".

A.1.3 Higher-order derivatives

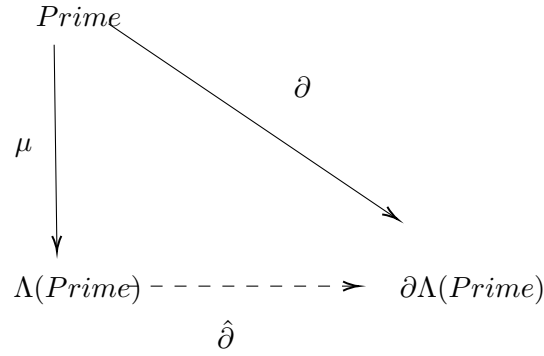
The mathematical meaning. It means that one is considering the second derivative, third derivative, and so on, of a function. We do not consider this for the moment.

B Intuition for the proof of the fact mentioned in BMP20

We here give a very naive intuition for the proof:

$Prime$ is the primal variables x , $\Lambda(Prime)$ is the backpropagator x^* , and they satisfy:

To be corrected



We want to show that $\hat{\partial}$ is a cartesian closed 2-functor. However I don't have enough time to dive in this perspective and it's quite away from my main topics.

C The Typing rules of original BMP20 approach

$$\begin{array}{c}
 \frac{}{\Gamma \vdash z : \mathbb{R}} \quad \frac{}{\Gamma, x^{!A} \vdash x : A} \quad \frac{\Gamma \vdash_{(z)} t : A \quad \Gamma \vdash_{(z)} u : B}{\Gamma \vdash_{(z)} \langle t, u \rangle : A \times B} \quad \frac{\Gamma \vdash u : A \times B \quad \Gamma, x^{!A}, y^{!B} \vdash_{(z)} t : C}{\Gamma \vdash_{(z)} t[\langle x^{!A}, y^{!B} \rangle := u] : C} \\
 \\
 \frac{\Gamma, x^{!A} \vdash t : B}{\Gamma \vdash \lambda x^{!A}. t : A \rightarrow B} \quad \frac{\Gamma \vdash t : A \rightarrow B \quad \Gamma \vdash u : A}{\Gamma \vdash tu : B} \quad \frac{\Gamma \vdash z t : \mathbb{R}^d}{\Gamma \vdash \lambda z^{\mathbb{R}}. t : \mathbb{R}^{\perp d}} \quad \frac{\Gamma \vdash t : \mathbb{R}^{\perp d} \quad \Gamma \vdash_{(z)} u : \mathbb{R}}{\Gamma \vdash_{(z)} tu : \mathbb{R}^d} \\
 \\
 \frac{\Gamma \vdash u : A \quad \Gamma, x^{!A} \vdash_{(z)} t : C}{\Gamma \vdash_{(z)} t[x^{!A} := u] : C} \quad \frac{\Gamma \vdash_{(z')} u : \mathbb{R} \quad \Gamma \vdash_z t : \mathbb{R}^d}{\Gamma \vdash_{(z')} t[z^{\mathbb{R}} := u] : \mathbb{R}^d} \quad \frac{\Gamma \vdash t_1 : \mathbb{R} \quad \dots \quad \Gamma \vdash t_k : \mathbb{R}}{\Gamma \vdash f(t_1, \dots, t_k) : \mathbb{R}} \quad \frac{r \in \mathbb{R}}{\Gamma \vdash \underline{r} : \mathbb{R}} \\
 \\
 \frac{\Gamma \vdash_{(z)} t : \mathbb{R} \quad \Gamma \vdash u : \mathbb{R}}{\Gamma \vdash_{(z)} t \cdot u : \mathbb{R}} \quad \frac{\Gamma \vdash t : \mathbb{R} \quad \Gamma \vdash_{(z)} u : \mathbb{R}}{\Gamma \vdash_{(z)} t \cdot u : \mathbb{R}} \quad \frac{}{\Gamma \vdash_z \underline{0} : \mathbb{R}^d} \quad \frac{\Gamma \vdash_{(z)} t : \mathbb{R}^d \quad \Gamma \vdash_{(z)} u : \mathbb{R}^d}{\Gamma \vdash_{(z)} t + u : \mathbb{R}^d}
 \end{array}$$

Figure 6: The typing rules. In the pairing and sum rules, either all three sequents have z , or none does.

D The Monadic translation and Reverse-mode wrapper of Kra+22

$\overleftarrow{\mathcal{D}}\{A\}$ Reverse mode on types

$$\begin{aligned}\overleftarrow{\mathcal{D}}\{A \times B\} &= \overleftarrow{\mathcal{D}}\{A\} \times \overleftarrow{\mathcal{D}}\{B\} \\ \overleftarrow{\mathcal{D}}\{A + B\} &= \overleftarrow{\mathcal{D}}\{A\} + \overleftarrow{\mathcal{D}}\{B\} \\ \overleftarrow{\mathcal{D}}\{A \rightarrow B\} &= \overleftarrow{\mathcal{D}}\{A\} \rightarrow M \overleftarrow{\mathcal{D}}\{B\} \\ \overleftarrow{\mathcal{D}}\{K\} &= K \\ \overleftarrow{\mathcal{D}}\{\mathbb{R}\} &= \mathbb{R} \times \text{Delta}\end{aligned}$$

$\overleftarrow{\mathcal{D}}\{e\}$ Reverse mode on expressions

Invariant: If $\Gamma \vdash e : A$, then $\overleftarrow{\mathcal{D}}\{\Gamma\} \vdash \overleftarrow{\mathcal{D}}\{e\} : M \overleftarrow{\mathcal{D}}\{A\}$

$$\begin{aligned}\overleftarrow{\mathcal{D}}\{x\} &= \text{pure } x \\ \overleftarrow{\mathcal{D}}\{\lambda x. e\} &= \text{pure } \lambda x. \overleftarrow{\mathcal{D}}\{e\} \\ \overleftarrow{\mathcal{D}}\{\text{let } x = e_1 \text{ in } e_2\} &= \text{do } \{x \leftarrow \overleftarrow{\mathcal{D}}\{e_1\}; \overleftarrow{\mathcal{D}}\{e_2\}\} \\ \overleftarrow{\mathcal{D}}\{e_1 e_2\} &= \text{do } \{f \leftarrow \overleftarrow{\mathcal{D}}\{e_1\}; x \leftarrow \overleftarrow{\mathcal{D}}\{e_2\}; f x\} \\ \overleftarrow{\mathcal{D}}\{\text{inl } e\} &= \text{do } \{x \leftarrow \overleftarrow{\mathcal{D}}\{e\}; \text{pure } (\text{inl } x)\} \\ \overleftarrow{\mathcal{D}}\{\text{inr } e\} &= \text{do } \{x \leftarrow \overleftarrow{\mathcal{D}}\{e\}; \text{pure } (\text{inr } x)\} \\ \overleftarrow{\mathcal{D}}\{\text{case } e_0 \text{ of inl } x_1 \rightarrow e_1, \text{ inr } x_2 \rightarrow e_2\} &= \text{do } \{x \leftarrow \overleftarrow{\mathcal{D}}\{e_0\}; \text{case } x \text{ of inl } x_1 \rightarrow \overleftarrow{\mathcal{D}}\{e_1\}, \text{ inr } x_2 \rightarrow \overleftarrow{\mathcal{D}}\{e_2\}\} \\ \overleftarrow{\mathcal{D}}\{(e_1, e_2)\} &= \text{do } \{x \leftarrow \overleftarrow{\mathcal{D}}\{e_1\}; y \leftarrow \overleftarrow{\mathcal{D}}\{e_2\}; \text{pure } (x, y)\} \\ \overleftarrow{\mathcal{D}}\{\text{fst } e\} &= \text{do } \{x \leftarrow \overleftarrow{\mathcal{D}}\{e\}; \text{pure } (\text{fst } x)\} \\ \overleftarrow{\mathcal{D}}\{\text{snd } e\} &= \text{do } \{x \leftarrow \overleftarrow{\mathcal{D}}\{e\}; \text{pure } (\text{snd } x)\} \\ \overleftarrow{\mathcal{D}}\{k\} &= \text{pure } k \\ \overleftarrow{\mathcal{D}}\{+z\} &= \text{pure } \lambda(x, y). \text{pure } (x +z y) \\ \overleftarrow{\mathcal{D}}\{r\} &= \text{pure } (r, \text{Zero}) \\ \overleftarrow{\mathcal{D}}\{+R\} &= \text{pure } \lambda((x, u_1), (y, u_2)). \\ &\quad \text{do } \{u_3 \leftarrow \text{deltaLet } (\text{Add } u_1 u_2); \\ &\quad \text{pure } (x + y, \text{Var } u_3)\} \\ \overleftarrow{\mathcal{D}}\{\times R\} &= \text{pure } \lambda((x, u_1), (y, u_2)). \\ &\quad \text{do } \{u_3 \leftarrow \text{deltaLet } (\text{Add } (\text{Scale } y u_1) (\text{Scale } x u_2)); \\ &\quad \text{pure } (x \times y, \text{Var } u_3)\}\end{aligned}$$

Figure 7: Monadic translation

Given $e : \mathbb{R}^a \rightarrow \mathbb{R}$
 and $\overleftarrow{\mathcal{D}}\{e\} : (\mathbb{R} \times \text{Delta})^a \rightarrow \mathcal{M}(\mathbb{R} \times \text{Delta})$ (as defined in Figure 9)
 we produce $\mathcal{R}\{e\} : (\mathbb{R}^a \times \delta\mathbb{R}) \rightarrow \delta\mathbb{R}^a$
 $\mathcal{R}\{e\} = \lambda((\dots, s_i, \dots), \delta t).$
 let $s_0 = (\dots, (s_i, \text{Var } i), \dots)$ **in**
 let $(_, u) = \text{runDelta } (a + 1) (\overleftarrow{\mathcal{D}}\{e\} s_0)$ **in**
 let $\text{finalMap} = \text{eval } \delta t \ u \ \text{emptyMap}$ **in**
 $(\dots, \text{lookupOrZero } i \ \text{finalMap}, \dots)$

Figure 8: Reverse-mode wrapper around $\overleftarrow{\mathcal{D}}$