

Functorial String Diagrams for Reverse-Mode Automatic Differentiation

Mario Alvarez-Picallo ✉ 

Programming Languages Laboratory, Huawei Research Centre, Cambridge, UK

Dan Ghica ✉ 

Department of Computer Science, University of Birmingham, UK

Programming Languages Laboratory, Huawei Research Centre, Cambridge, UK

David Sprunger ✉ 

Department of Computer Science, University of Birmingham, UK

Fabio Zanasi ✉ 

Department of Computer Science, University College London, UK

Abstract

We formulate a reverse-mode automatic differentiation (RAD) algorithm for (applied) simply typed lambda calculus in the style of Pearlmutter and Siskind [27], using the graphical formalism of string diagrams. Thanks to string diagram rewriting, we are able to formally prove for the first time the soundness of such an algorithm. Our approach requires developing a calculus of string diagrams with hierarchical features in the spirit of functorial boxes, in order to model closed monoidal (and cartesian closed) structure. To give an efficient yet principled implementation of the RAD algorithm, we use *foliations* of our hierarchical string diagrams.

2012 ACM Subject Classification Mathematics of computing → Automatic differentiation; Theory of computation → Categorical semantics

Keywords and phrases string diagrams, automatic differentiation, hierarchical hypergraphs

Digital Object Identifier 10.4230/LIPIcs.CSL.2023.6

Funding The last three authors acknowledge support from EPSRC grant EP/V002376/1.

1 Introduction

Interest in incorporating differential structure to programming languages has grown with their use in machine learning, especially to support gradient-based training algorithms [4]. A central piece of these algorithms is reverse-mode automatic differentiation (RAD, also called backpropagation), which has proven to be an efficient way to compute gradients. For programs constructed entirely of differentiable operations on (tuples of) real numbers, automatic differentiation amounts to a mechanisation of the chain rule from (finite dimensional) multivariable calculus. However, in many programming languages, programs can themselves consume and produce other programs. Since spaces of functions on non-trivial real vector spaces are infinite dimensional, the differentiation of these higher-order programs cannot rely on the same justifications for correctness. Moreover, a notion of differentiation for higher-order programs can be useful even in situations where it is not necessary. For example, mapping a function over a list can always be expressed by an equivalent first-order program, but rewriting a program to fully remove higher-order functions is often not as convenient or efficient as a higher-order approach would be.

In this paper, we propose a differentiation algorithm for higher-order code which incorporates the fine-grained structure induced by defining and reusing programs within programs. In a nutshell, our algorithm extends every closure in the code with a back-propagator which computes the gradient of the original function along a given vector. It



© Mario Alvarez-Picallo, Dan Ghica, David Sprunger, and Fabio Zanasi;
licensed under Creative Commons License CC-BY 4.0

31st EACSL Annual Conference on Computer Science Logic (CSL 2023).

Editors: Bartek Klin and Elaine Pimentel; Article No. 6; pp. 6:1–6:20

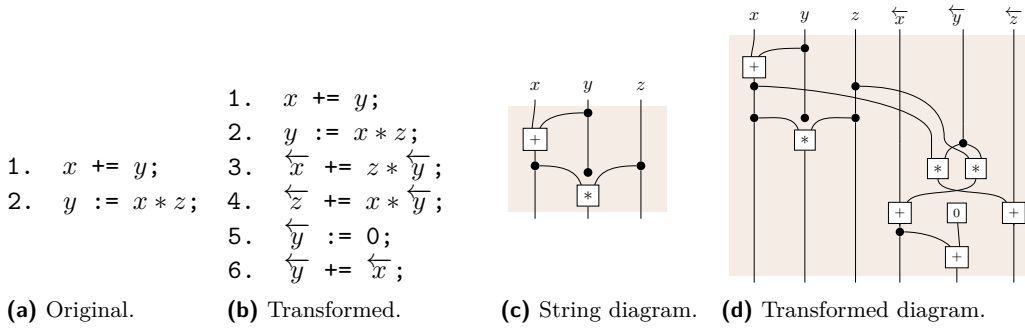
Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

6:2 Functorial String Diagrams for RAD

accomplishes this by executing a reversed version of the original computation graph. In order to differentiate higher-order functions, we also collect the partial derivatives of a function’s output with respect to any captured variables/terms in its environment and pass those as the tangent value corresponding to an input of function type – in a similar style to Pearlmutter and Siskind’s automatic differentiation algorithm [27].



■ **Figure 1** Approaches to RAD: program transformation and graphical.

In a typical program transformation approach to reverse-mode AD, each line of a program is paired with a small program computing that line’s derivative. These derivatives are then assembled in the reverse order of the corresponding lines to obtain a transformed program which computes the original program (the “primal”) together with its derivative. An example two-line primal program and its transformed version are shown in Figure 1a and Figure 1b. Note the first two lines of the original program are repeated (the “forward pass”). Lines 3–5 compute the derivative of line 2 and line 6 computes the the derivative of line 1, together forming the “reverse pass”.

A good explanation of such a transformation should be detailed enough to serve as a complete specification for implementation and verification while also conveying the high-level ideas to facilitate necessary extensions. For example, the transformation of [27] assumes the program has been preprocessed to straight-line single-assignment code with unary and binary operations only. These assumptions rule out in-place assignments like line 1, so without high-level guidance one may not discover that destructive assignments like line 2 must have a zeroing line (line 5) in their reverse pass while nondestructive assignments should not.

The main novelty of our approach is the use of *category theory* to provide a firm semantic foundation to the algorithm. Our categorical environment features both the axioms of *cartesian closed categories*, modelling the operations of a generic functional language, and of *reverse differential categories* [12], accounting for first-order differentiability. We extract a detailed RAD transformation from the reverse differential operator; the axioms governing that operator provide a flexible high-level description one could use, for example, to understand why destructive and nondestructive assignment must be treated differently.

Roughly speaking, programs in these functional languages correspond with morphisms in these categories. To facilitate automated manipulation of these morphisms, we represent them as *string diagrams*, which are a graphical, yet completely formal language to represent morphisms [29]. For example, the program of Figure 1a corresponds with the string diagram of Figure 1c and its RAD transformed version (Figure 1b) corresponds to the string diagram of Figure 1d. To represent higher-order functions, we rely on a more sophisticated string diagram, namely **hierarchical string diagrams**, which in turn use *functorial boxes* [25].

A second ingredient that we need to introduce is the notion of **foliation**. A foliation is a “maximally spread out” representation of a string diagram; **leaves** in these foliations roughly correspond to lines in a program. This enables us to reason by *induction* on diagrams.

We formulate our algorithm as a rewriting system which, when applied to a foliation of a string diagram, produces its reverse-mode derivative. Thanks to the underlying reverse differential structure, using equational reasoning on string diagrams we are able to formulate and prove correctness of the algorithm (Theorem 17).

At a more fundamental level, these developments show how the dual nature of string diagrams – which may be interpreted both as (hyper)graphs and as a formal syntax – offers new perspective in the study of AD. On the one hand, string diagrams are akin to lower-level structures standardly found in AD implementations, such as computation graphs [1], and can be manipulated combinatorially. At the same time, their syntactic specification and algebraic properties enable methodologies typical of more abstract formalism, such as the approaches based on functional programming [13, 27]. Particularly, the use of induction, rewriting, and reverse derivative categories provide a basis for arguing soundness we do not see in computation graphs alone. In this sense, our approach ideally acts as a bridge between different formalisms to reason on AD algorithms.

Synopsis. In Section 2 we build towards the definition of hierarchical string diagrams. In Section 3 we develop the tools necessary to compute with these diagrams: foliations, hierarchical hypergraphs, and reverse differentiation. In Section 4 we present our algorithm for automatic differentiation and prove its correctness. Finally in Sections 5 and 6, we discuss related work and future work.

2 Hierarchical string diagrams

In this section we give a diagrammatic introduction to the basics of monoidal closed categories. String diagrams in this work are to be read from top to bottom.

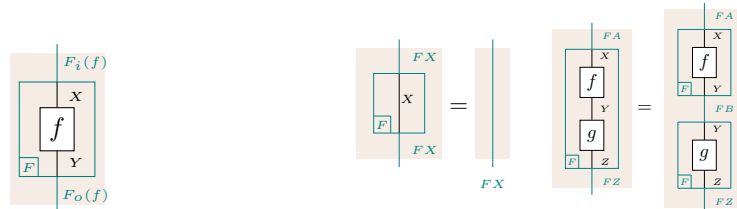
2.1 Functorial string diagrams

Given a category, we denote the identity on an object X by id_X , a morphism $f: X \rightarrow Y$

by $\begin{array}{c} x \\ \boxed{f} \\ y \end{array}$, and write $\begin{array}{c} x \\ \boxed{f} \\ \boxed{g} \\ z \end{array}$ for the composition $f;g$ of morphisms $f: X \rightarrow Y$ and $g: Y \rightarrow Z$. Note

the string diagrammatic notation neatly absorbs associativity and identity of composition.

Following [25], we extend string diagrams with *labelled frames* which indicate mappings between categories. The application of a mapping F to a morphism f can be seen in Figure 2a. Often such a mapping is a *functor*, respecting composition and identity as depicted in Figure 2b. We write $1_{\mathcal{C}}$ for the identity functor on category \mathcal{C} .



(a) Labeled frame.

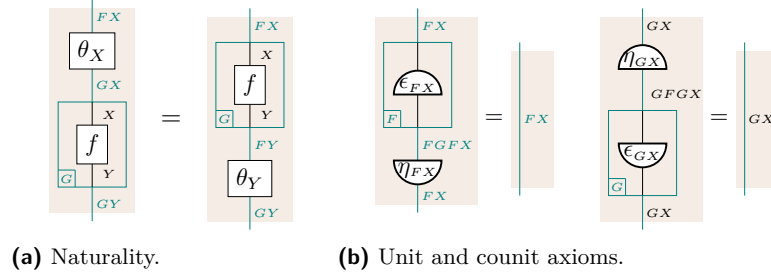
(b) Functor axioms.

Figure 2 Labeled frames and functors.

6:4 Functorial String Diagrams for RAD

A natural transformation from a functor F to a parallel functor G is an object-indexed family of morphisms $\theta_A : FX \rightarrow GX$ satisfying the property depicted in this diagrammatic language in Figure 3a.

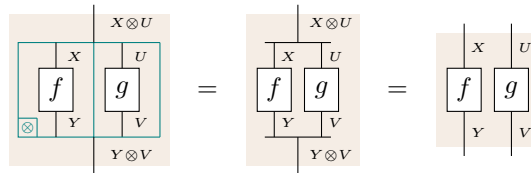
We may also express an *adjunction* between functors $F : \mathcal{D} \rightarrow \mathcal{C}$ (left adjoint) and $G : \mathcal{C} \rightarrow \mathcal{D}$ (right adjoint), via natural transformations $\epsilon : FG \rightarrow 1_{\mathcal{C}}$ and $\eta : 1_{\mathcal{D}} \rightarrow GF$, which satisfy unit and counit axioms found in Figure 3b. We will write the counit of an adjunction as a lower semicircle $\begin{array}{c} FGX \\ \epsilon_X \\ X \end{array}$, the unit as an upper semicircle $\begin{array}{c} Y \\ \eta_Y \\ GFY \end{array}$, and omit the label when the map is clear from context.



■ **Figure 3** String diagrams for natural transformations and adjunctions.

2.2 String diagrams for monoidal categories

The diagrammatic notation can be generalised to bifunctors by drawing a two-place frame. One bifunctor that plays a special role in string diagrams is the *tensor product* or *monoidal product*, in particular when it is *strict*. The strict tensor is represented diagrammatically as:







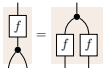
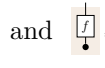
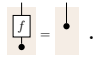



The diagram above contains three representations. In the first one we see tensor as a bifunctor, with the two separate boxes indicating the two arguments of the bifunctor. The second one is special notation for the tensor, essentially hiding the functorial box and using a graphical convention (the horizontal line) to represent the “unravelling” of the tensored-labelled stem into components. Finally, the third is special notation for strict monoidal tensor, in which the tensor $X \otimes U$ is represented as the list of its components $[X, U]$.

The category is *symmetric* monoidal when there is a “symmetry” natural transformation $\gamma_{X,Y} : X \otimes Y \rightarrow Y \otimes X$ which is involutive, i.e. $\gamma_{X,Y}; \gamma_{Y,X} = \text{id}_X \otimes \text{id}_Y$. Diagrammatically, we represent symmetry by $\begin{array}{c} \gamma \\ \text{---} \\ \text{---} \end{array} := \begin{array}{c} \text{---} \\ \text{---} \end{array}$. As a consequence of the naturality of γ , we can

“push” morphisms through symmetry: $\begin{array}{c} \gamma \\ \text{---} \\ \text{---} \end{array} \begin{array}{c} f \\ \text{---} \\ \text{---} \end{array} = \begin{array}{c} \text{---} \\ \text{---} \end{array} \begin{array}{c} g \\ \text{---} \\ \text{---} \end{array}$.

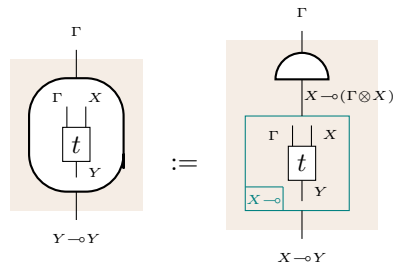
A *cartesian (monoidal) category* is a monoidal category where the tensor operation and unit are given by categorical products and the terminal object. One standard recipe to enforce this condition (see e.g. [19, Theorem 6.13]) on a monoidal category is to add monoidal natural transformations $\delta_X : X \rightarrow X \otimes X$ (contraction) and $\omega_X : X \rightarrow I$ (weakening) making I terminal, making every object into a cocommutative comonoid, and

interacting nicely with the product structure, i.e. $\delta_{X \otimes Y} = (\delta_X \otimes \delta_Y); (\text{id}_X \otimes \gamma_{X,Y} \otimes \text{id}_Y)$. We represent contraction as  and weakening as . The cocommutative comonoid equations include  =  and  = . Copying and discarding are both consequences of naturality, i.e.  =  and  = .

2.3 Hierarchical string diagrams

Monoidal closed categories and cartesian closed categories are fundamental in the construction of categorical models for the linear and simply-typed λ -calculus, respectively [6]. A *symmetric monoidal closed category* is a symmetric monoidal category where for every object Y , the (endo)functor $F_Y(X) = X \otimes Y$ has a right adjoint $G_Y(X) = X \multimap Y$. The equations defining this adjunction are presented diagrammatically in Appendix A.

We will not be using these here, but rather a hom-set presentation of the adjunction, consisting of a natural bijection between $\mathcal{C}(\Gamma \otimes X, Y)$ and $\mathcal{C}(\Gamma, X \multimap Y)$. This bijection is known as “currying”, and is a more germane presentation for the λ -calculus. We define *abstraction*, the composition of the unit of the adjunction with the functorial box for G , as syntactic sugar denoted by a plain box with rounded corners.



This structure for abstraction yields our notion of a **hierarchical string diagram**, which is to say a string diagram which may contain other string diagrams in these boxes.

A cartesian monoidal category which is also a monoidal closed category is called a *cartesian closed category*. In the sequel we will restrict ourselves to cartesian closed categories, and so we will write \times for \otimes and \Rightarrow for \multimap .

For computational purposes, it is important to have a data structure efficiently interpreting string diagrams. For string diagrams in symmetric monoidal categories, such a data structure is provided by hypergraphs with interfaces [8]. This can be extended to our closed context with *hierarchical hypergraphs* [2]. We also point out that foliations for hierarchical string diagrams (see Section 3.2) can be found efficiently by interpreting the string diagram as a hypergraph, then topologically sorting the edges.

3 Foliations and reverse differentiation

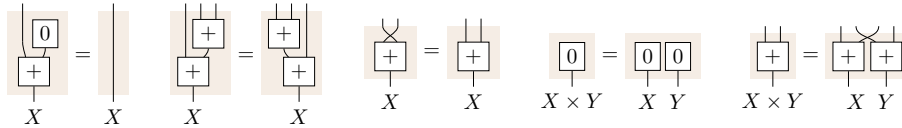
Next we focus on cartesian closed categories freely generated from a signature. In order to define differential structure, these signatures must include the vector space-like structure of (cartesian closed) left additive categories [7]. For our developments, we are especially interested in using *foliations*, special presentations of morphisms in these categories which enable the use of inductive definitions and proofs.

3.1 Freely generated categories

► **Definition 1.** A signature $(\Sigma_0, \Sigma_1, i, o)$ consists of a set Σ_0 of types, a set Σ_1 of operations, and functions $i, o : \Sigma_1 \rightarrow \Sigma_0^*$ giving input and output types for each operation.

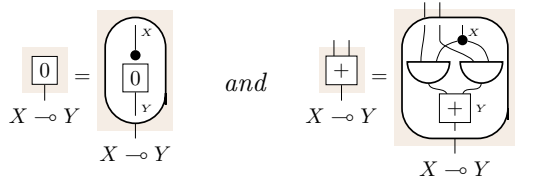
Derivatives are usually defined to be the linear map best approximating a function near a point. For us, this notion is provided by left additive categories, which introduce a monoid structure on homsets generalizing vector space addition.

► **Definition 2 ([12]).** A cartesian category \mathcal{C} is cartesian left additive if each object X in \mathcal{C} is a commutative monoid, meaning there are morphisms $0_X : 1 \rightarrow X$ and $+_X : X \times X \rightarrow X$ satisfying the first three equations depicted below. Furthermore, these monoids are required to be compatible with the cartesian structure in the sense of the last two equations.



It is equivalent to say that every hom-set $\mathcal{C}(X, Y)$ is a commutative monoid, compatible with the cartesian structure; a full account of this notion can be found in [12]. Note that we do not assume any interaction between the additive structure and the comonoid structure given by counit and comultiplication; e.g. equations such as do not hold in most models.

► **Definition 3 ([7, 1.4]).** A cartesian closed left additive category is both a cartesian left additive category and a cartesian closed category which respects the additive structure:



Now we can give the setting for our automatic differentiation algorithm.




► **Definition 4.** Suppose Σ is a signature. We let \mathcal{A}_Σ be the free (strict) cartesian left additive category generated by Σ , and we let \mathcal{C}_Σ be the free (strict) cartesian closed left additive category generated by Σ . Naturally, \mathcal{A}_Σ is included in \mathcal{C}_Σ .

The construction of these free categories is standard.

3.2 Foliations

Freely generated categories provide inductive structures with which we can describe all objects and morphisms. Being able to factorise morphisms gives us access to *foliations*, which allow us to treat morphisms as almost syntactic objects that we can perform induction on. Foliation-like objects, such as the more general polygraphs, are well-known in the literature, see for example [11, 18, 34]. Of particular relevance is the use of foliations in [35] to enable the programmatic manipulation of string diagrams. We fix here the relevant definitions as they relate to hierarchical string diagrams.

► **Definition 5.** Fix a set of string diagrams Σ_E , which we call atomic.¹ A foliation (with respect to Σ_E) is a sequential composition of string diagrams, which we call leaves. A leaf consists of an atomic morphism or an abstraction, tensored with any number of identities on either side. A maximal foliation is a foliation comprising only leaves. A maximal hierarchical foliation is a maximal foliation which is either abstraction free, or in which all abstracted diagrams are also maximal hierarchical foliations.

For instance, the maximal foliations of  are  and , if f and g are atomic.

The following is an obvious generalisation of a folklore theorem.

► **Lemma 6.** Any hierarchical string diagram can be written as a (non-unique) maximal hierarchical foliation.

The proof is straightforward: intuitively, whenever two terms are “level” in a diagram one of them can be “shifted” using identities, then tensors and compositions can be reorganised using the functoriality of the tensor.

Foliations are inductive, allowing syntactic transformations defined recursively. Then instead of defining “big” rules for general sequential and parallel composites, we only need “small” rules for composing a morphism with a leaf, which is the format presenting the transformation in Figure 8. This also makes for simpler inductive proofs about the foliations.


3.3 Reverse differential categories

Finally, we present a graphical treatment of basic reverse-mode differentiation without abstraction or function evaluation. The *reverse-mode derivative* of a function $f : \mathcal{R}^m \rightarrow \mathcal{R}^n$ is the function $(Jf)^\top : \mathcal{R}^m \rightarrow (\mathcal{R}^n \rightarrow \mathcal{R}^m)$ taking a base point in the domain of the function to the (linear map represented by the) transpose of the Jacobian of the function at that point. We denote the reverse mode derivative of a function by Rf .

► **Example 7.** Consider binary multiplication, $m : \mathcal{R}^2 \rightarrow \mathcal{R}$ given by $m(x, y) = x \cdot y$. The Jacobian of this function at (x_0, y_0) is $[y_0 \ x_0]$. Recall this means that given a small change in the inputs $[\Delta x \ \Delta y]$ to m , the output will change by approximately $[y_0 \ x_0] [\Delta x \ \Delta y] = \Delta x \cdot y_0 + x_0 \cdot \Delta y$. Thus, the reverse-mode derivative of m is the function $Rm : \mathcal{R}^2 \rightarrow (\mathcal{R} \rightarrow \mathcal{R}^2)$ given by $(Rm)(x_0, y_0) = [\Delta z \mapsto (y_0 \cdot \Delta z, x_0 \cdot \Delta z)]$.

The idea of this “reverse-mode derivative” is captured in the formalism of reverse differential categories [12].

► **Definition 8** ([12, Def. 13]). A reverse differential category, or reverse derivative category, (RDC) is a cartesian left additive category endowed with a combinator R sending each morphism $f : X \rightarrow Y$ to a morphism $R[f] : X \times Y \rightarrow X$. This combinator satisfies seven axioms (RD1-7) which can be found in both equational and diagrammatic form in Appendix B.

We will introduce the axioms of the reverse derivative combinator necessary for understanding our algorithm gradually. We will use the a pink frame  to denote the reverse derivative Rf of a morphism f in diagrammatic form. Note we are using the morphism frame notation for a *non-functorial* operation on morphisms. Despite this, this operator does have interesting compositional properties, as shown by axioms RD4 and RD5.

¹ Typically, we will take Σ_E to be the morphisms of a signature, together with some set of structural morphisms in a category.

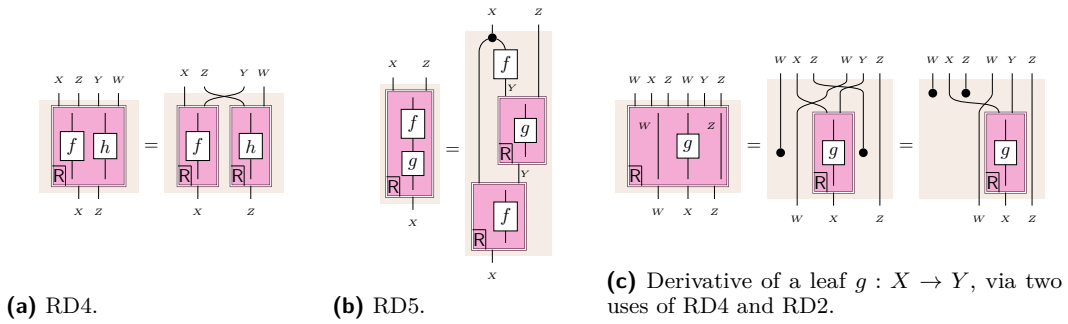


Figure 4 RDC axioms as string diagrams.

Next we consider an important class of morphisms: the linear morphisms.

► **Definition 9.** A morphism $f : X \rightarrow Y$ in a reverse differential category is linear whenever there is a morphism $g : Y \rightarrow X$ such that $R[f] = \omega_X \times g$ and $R[g] = \omega_Y \times f$.

Note that if f is linear, then so is g . We call g the dual of f and denote it by f^\dagger .

The subcategory of linear maps in an RDC is a dagger category [12, Prop. 24].

► **Lemma 10.** When f and g are linear maps in a reverse differential category, so are $f; g$ and $f \times g$ and their duals are $(f; g)^\dagger = g^\dagger; f^\dagger$ and $(f \times g)^\dagger = f^\dagger \times g^\dagger$.

The axioms of reverse differential categories require the structural morphisms (id_X , $\gamma_{X,Y}$, δ_X , ω_X , 0_X , and $+_X$) to be linear, and further require that 0_X is dual to ω_X , $+_X$ is dual to δ_X , $\gamma_{X,Y}$ is dual to $\gamma_{Y,X}$, and id_X is dual to itself.

In order to equip categories generated from a signature with reverse differential structure, it is sufficient to designate a suitable derivative for each of the basic operations by the following lemma.

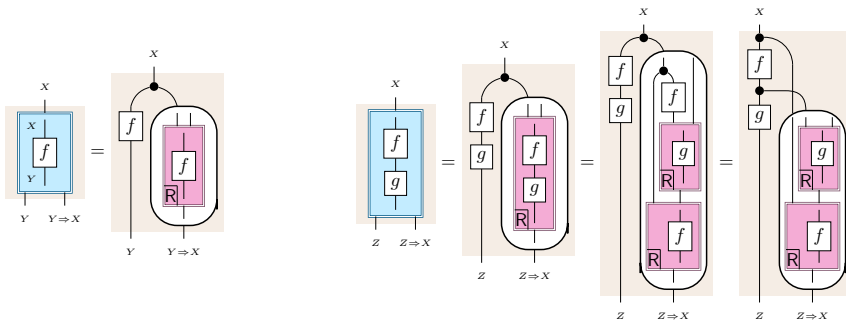
► **Lemma 11.** Suppose Σ is a signature and suppose that for every operation $g : X \rightarrow Y$ in Σ_1 , there is a well-defined reverse derivative $R[g]$ morphism in \mathcal{A}_Σ satisfying RD2, RD6, and RD7. Then \mathcal{A}_Σ is a reverse differential category.

In such a case, we will call Σ a differentiable signature.

It is an exercise for the reader to check that with the reverse-mode derivative of multiplication defined in Example 7, the string diagram of Figure 1d is the tupling of the string diagram of Figure 1c with its reverse-mode derivative. In this exercise, one will find that the zeroing of \overleftarrow{y} is the dual of the discard of the destructive assignment. Thus, that this requires a zeroing of the corresponding reverse-pass variable (while in-place operations do not) is a consequence of the RDC axioms.

3.4 Optimizing RAD string diagrams

Reverse differential categories are defined without closed structure to maximize generality. However, in a closed setting, we can abstract the differential component of input to the reverse derivative. With this, we can more precisely state the goal of this work: we seek an “adjoint transformation” taking $f : X \rightarrow Y$ to $Af : X \rightarrow Y \times (Y \rightrightarrows X)$, which augments the result of f with a function representing its reverse-mode derivative at the given input. We depict this transformation in Figure 5a.



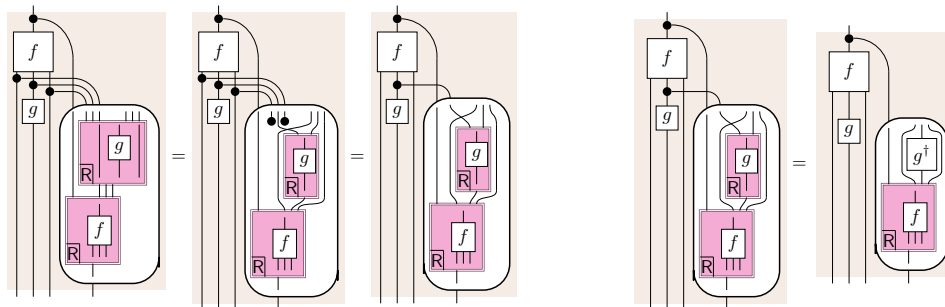
(a) Adjoint transformation. (b) Adjoint transformation applied to sequential composition.

■ **Figure 5** Basics of the adjoint transformation.

The RDC axioms, particularly RD5, tell us how to compute this adjoint transformation when f can be decomposed as a sequential composite, as shown in Figure 5b. This immediately suggests important, common optimizations in practical implementations.

In proceeding from the third diagram to the fourth, some computation is saved (f is used only once on the right) at the cost of having to transmit an extra intermediate result to the closure giving the derivative. This is a common time-space tradeoff in backpropagation where usually space is sacrificed.

Next, if the second layer of computation is a leaf in a foliation, we can make this diagram even more precise. Substituting the reverse derivative of a leaf from Figure 4c into the diagram in Figure 5b, we obtain Figure 6a. This expresses space savings common in all practical implementations: the whole state of the computation does not need to be saved at each step, only the values which are used in the next operation.



(a) Adjoint transformation on a leaf.

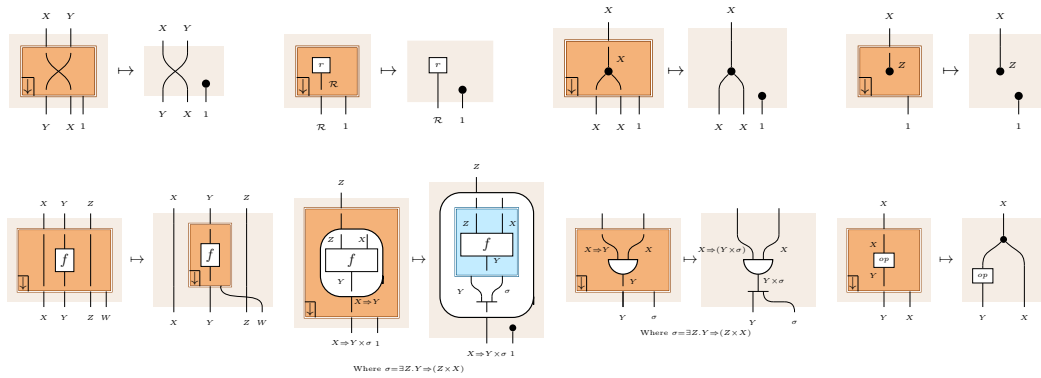
(b) Adjoint transformation on a linear leaf.

■ **Figure 6** Adjoint transformation on (linear) leaves.

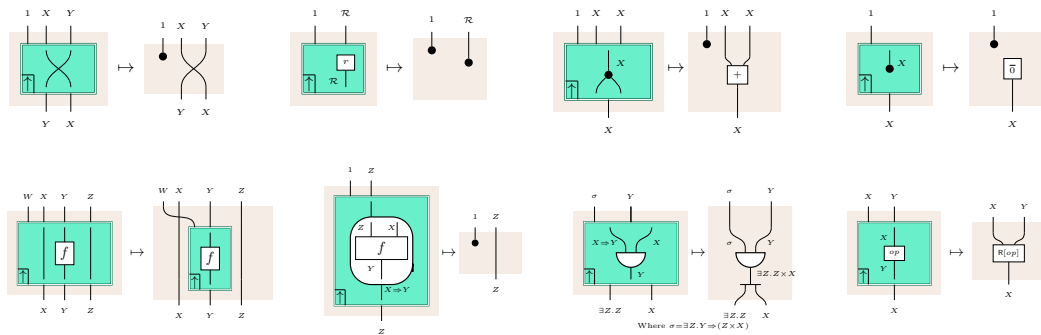
Finally, if the operation in a leaf is linear, we can simplify to Figure 6b. This saves even more space – no inputs to this step need be saved since the derivatives of linear operations do not depend on the base point.

Section 3.4, intermediate values computed during the forward pass are stored and passed along to the diagram corresponding to the reverse pass, shown in Fig. 7a as a bundle of type Γ flowing from the forward pass into the backpropagator, which we refer to as the *context* wire.

The adjoint transformation corresponds to the “reverse transform” of [27, p. 26]. The definition of this reverse transform is shown in Figure 7b. This definition references two other per-line transformations ϕ and ρ , which correspond to our forward (ϕ) and reverse (ρ) passes. The ϕ transformation generally keeps the same program shape, occasionally binding new variables; analogously our forward pass occasionally saves intermediate values for the backpropagator. Inside the binding, Pearlmutter and Siskind’s reverse transformation zeroes many variables (since they assume the program has been preprocessed to single-assignment), then ρ -transforms the original code, analogous to our reverse pass.



(a) Rewrites defining the forward pass.



(b) Rewrites defining the reverse pass.

■ **Figure 8** Forward and reverse pass rewrites.

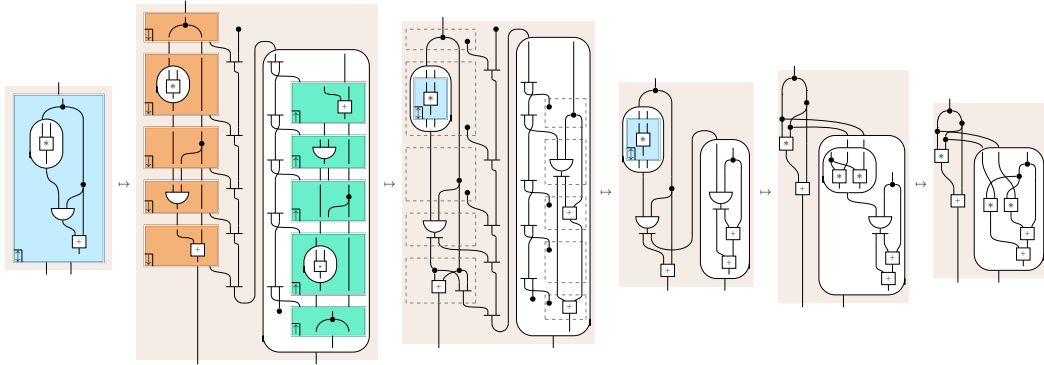
The first four rules (top row) of the forward and reverse pass follow the logic introduced in Section 3.4: linear morphisms are simply replaced by their dual in the reverse pass – observe that, since the reverse derivative of a linear diagram does not depend on its base point, these rules only generate a “dummy” unit value to be added to the context wire. We have already seen the logic for the last rule on the bottom row as well: for a generic nonlinear basic operation, the forward pass needs to save the basepoint and the reverse pass uses that for its reverse-mode derivative.

6:12 Functorial String Diagrams for RAD

► **Remark 12.** We note that each copy node in the primal code is replaced by an addition in the reverse pass, adding up all the sensitivities corresponding to each use of a variable before propagating them further up the computation graph. This prevents the *fanout problem* and ensures efficiency without requiring additional optimization passes like *linear factoring* used by [10] or sophisticated, quasi-symbolic representations of operations as in [22].

The rules for abstraction (left bottom row) and evaluation (middle bottom row) are harder to back up with compelling intuitions. For abstraction, the diagram enclosed by the bubble is recursively transformed using the blue rule in the forward pass – that is, any abstraction in the primal diagram is replaced by a new abstraction that computes the adjoint of the original one. Then, when this function is evaluated in the primal diagram, the forward pass gets the result of the adjoint application – both the result of the original abstraction and a backpropagator which is not used in the forward pass but is set aside for the reverse pass.

When rewriting an application node, the reverse pass applies the backpropagator produced by the forward pass. This backpropagator in turn produces a wire for every operand of the body of the original abstraction. The abstraction rule in the reverse pass then expects the sensitivity of an abstraction to consist of a bundle of wires corresponding to the sensitivities of each captured wire.



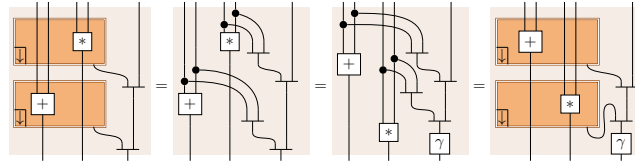
■ **Figure 9** Differentiating a simple polynomial.

As an example, consider the program `let mul z = y * z in mul y + y`, which computes the polynomial $y \mapsto y^2 + y$. In Fig. 9, we show the result of applying the adjoint transformation to this diagram. We assume that the signature that generates our category includes a multiplication³ morphism $\boxed{\cdot}$ whose corresponding reverse derivative is given by $\boxed{+}$ (which is simply the gradient of the product familiar from basic calculus). The resulting backpropagator, when applied to input 1, gives the correct derivative of the original polynomial.

► **Lemma 13.** *The adjoint transformation defined by Fig. 8a and Fig. 8b is independent of the choice of foliation, up to a permutation of the context wire.*

Proof (Sketch). By induction on the length of the foliation. It can be easily checked that every pair of such rules commutes, modulo a permutation of the wires that are propagated from the forward to the reverse pass. For a concrete example, consider the two rewrites stemming from two different foliations of the same graph in Fig. 10, and note that they differ only in the obvious permutation morphism $\gamma : X \times (Y \times \Gamma) \rightarrow Y \times (X \times \Gamma)$. ◀

³ Multiplication here should be understood as multiplication of real numbers, and not be confused with the product structure of our category.

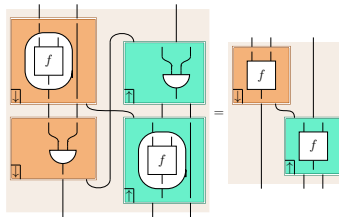


■ **Figure 10** Two rewrite sequences stemming from two distinct foliations of the same graph.

► **Remark 14.** The proof above, although simple, illustrates a proof method that is made possible by using string diagrams: induction on the length of the *foliation* of the diagram (Def. 5). This proof method also benefits from absence of names and all related bureaucratic concerns (free vs. bound variables, alpha equivalence, capture-avoiding substitution).

4.2 Correctness

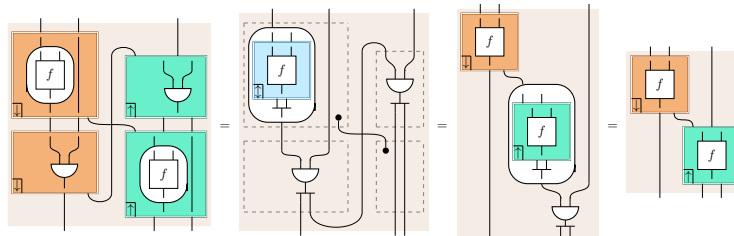
Our proof of correctness proceeds in two steps. First, we prove that our AD transformation is compatible with Beta reduction: whenever two diagrams are equivalent modulo Beta reduction, then so are their adjoints. Then, we show that the AD transformation is correct for diagrams featuring only first-order nodes (featuring no abstractions or applications).



■ **Figure 11** AD is compatible with Beta-reduction.

► **Lemma 15.** *The rewriting rules in Fig. 8 are compatible with beta reduction (The equation in Fig. 11 holds).*

Proof. The proof proceeds by straightforward application of the rewrite rules. We provide the calculation in full in Fig. 12. ◀



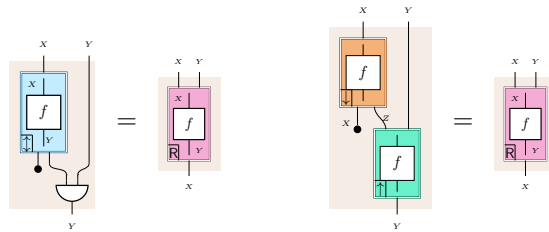
■ **Figure 12** Beta-soundness of AD.

6:14 Functorial String Diagrams for RAD

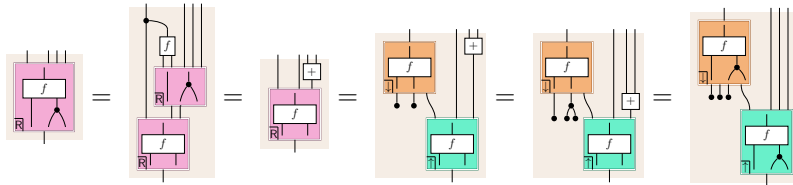
► **Lemma 16.** For every \boxed{f} whose operands and results are all of a first-order type, there is a Beta-equivalent \boxed{f} that contains no instances of abstraction or evaluation and whose every node has all first-order inputs and outputs.

Proof. Since our graphical language is simply-typed, evaluation of \boxed{f} is guaranteed to terminate, following an argument similar to the proofs of strong normalisation for the simply-typed λ -calculus (such as the one in [16, Chapter 6]), or for interaction nets (e.g. [24]). Such a normal form cannot contain any redexes, so any abstraction or eval node must be connected to an operand or a result wire, which cannot happen as these have all first-order types. ◀

► **Theorem 17.** For every \boxed{f} whose operands and results are all first-order, the following (equivalent) equations hold:



Proof. Applying Lemma 16 and Lemma 15, it suffices to consider the case where \boxed{f} contains no instances of application or evaluation. Applying the rewrite rule in Fig. 7a shows the equivalence of the two equations above. The result then follows by induction on the foliation of the diagram \boxed{f} . We show the case for contraction in full, with the other cases being similar.



5 Related work

5.1 String diagrams

Cartesian closed categories have been thoroughly studied in the context of logic and type theory, because of the well-known correspondence of their internal language with λ -calculus and intuitionistic logic [32]. The *linear* version of this triad involves monoidal rather than cartesian categories, but also proof nets, and linear logic, as indicated already in the original paper [15]. [25] provides the foundation on which we build our language of string diagrams, noting that all the basic ingredients are already there.

The route of using an enhancement of the monoidal closed structure with additional properties to control sharing is fruitful and has been employed many times. For example it is found in [9], where the manipulation of variables endowed with algebraic theories is modelled as a cartesian structure on the top of a linear structure, or in [14] to specify multiplexers and demultiplexers in high-level synthesis.

Our approach to string diagrams shares similarities with the formalisms of sharing graphs for describing λ -calculus computations [23]. The main difference is that string diagrams can be manipulated as a syntax, whereas sharing graphs are usually studied as combinatorial objects. Unlike syntax, reasoning about graphs algebraically requires a higher degree of technical sophistication [17]. Finally, sharing graphs are typically used to study low-level computational models for functional languages, in particular quantitative resource models [26], whereas our approach is more focussed on equational reasoning and rewriting.

Monoidal closed categories extend not only to cartesian closed categories, but also to \star -autonomous categories. This second variation is very much relevant to the study of multiplicative linear logic and it has been extensively studied in terms of proof nets. Our graphical calculus is essentially different from proof nets. The grammar of generating morphism does not stem from a sequent calculus, and we capture the intended semantics via equations rather than a correctness criterion. But the connection might be made precise relying on the existing translations between proof nets and string diagrams [20, 30].

Baez and Stay [3] propose an intriguing graphical innovation for monoidal closed categories, a so-called *clasp* operator on stems. The exponential type is represented using the clasp, and much like in our own language, a *bubble* is used to represent currying. It is unclear to us how to represent higher-order objects like $(A \multimap B) \multimap C$ using the clasp. However, we think it preserves an appealing visual parallel between monoidal closed and compact closed structures.

Although not presented explicitly as a string diagram language, the treatment of closures in [28] is related in methodology to our work, although the use of *partially-traced partially-closed premonoidal categories* as the categorical setting, in order to accommodate for effects, is significantly different than our cartesian closed categorical language.

5.2 Automatic differentiation

Our AD algorithm is an adaptation of [27]. Beyond the presentation based on string diagrams, the main differences are that our algorithm applies to simply-typed, recursion-free code and it acts as a source-to-source transformation, lacking the reflection features that enabled higher-order differentiation in the original work. We chose to focus on this algorithm for a few reasons: first, reverse-mode AD is both more immediately useful (see [4] for a comparison of both approaches) and harder to implement and prove correct than forward-mode AD. Second, it is to our knowledge the first published algorithm for performing reverse-mode AD on higher-order code, it forms the basis of a number of efficient implementations [5, 31] and does not require more complex features, unlike e.g. [36] which makes use of mutable state and continuations or [10] which relies on a limited form of continuations to encode dual spaces.

A wave of recent research has also tackled the issues of correctness in automatic differentiation. Notably, [10] and [33] provide correct reverse-mode AD algorithms capable of handling closures. Unlike the first work, however, our algorithm is purely functional and, while the second one can correctly differentiate terms with higher-order inputs and outputs, it does so by using a more expensive representation of tangents of function spaces. The main contribution of our approach, however, is the simplicity of the involved proofs thanks to our diagrammatic notation which we believe improves on the readability of the original paper [27] and the denser proofs in newer literature [10, 21, 33].

6 Conclusion and further work

In this paper we have presented a recipe for provably correct reverse automatic differentiation built around a hierarchical calculus of string diagrams. As we have seen, the string diagram presentation simplifies much of the bookkeeping of variables a term calculus would require, which makes a complicated algorithm more readable. More importantly, the new perspective offered by string diagrams and in particular the presentation of terms as *foliations* opens the door for new and useful proof techniques.

We have not discussed implementation matters, yet these are of the essence. This is a practical algorithm and it can be incorporated into real-life compilers for real-life programming languages. We surmise that the improved perspective on AD that string diagrams offers will help handle other challenging features of real-life languages, such as effects and recursion. This work is ongoing.

References

- 1 Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. TensorFlow: a system for large-scale machine learning. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, pages 265–283, 2016.
- 2 Mario Alvarez-Picallo, Dan Ghica, David Sprunger, and Fabio Zanasi. Rewriting for monoidal closed categories. In *7th International Conference on Formal Structures for Computation and Deduction (FSCD 2022)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2022.
- 3 John Baez and Mike Stay. Physics, topology, logic and computation: a rosetta stone. In *New structures for physics*, pages 95–172. Springer, 2010. doi:10.1007/978-3-642-12821-9_2.
- 4 Atilim Gunes Baydin, Barak A. Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic differentiation in machine learning: a survey. *J. Mach. Learn. Res.*, 18:153:1–153:43, 2017. URL: <http://jmlr.org/papers/v18/17-468.html>.
- 5 Atilim Günes Baydin, Barak A. Pearlmutter, and Jeffrey Mark Siskind. Diffsharp: An AD library for .net languages. *CoRR*, abs/1611.03423, 2016. arXiv:1611.03423.
- 6 Nick Benton, Gavin Bierman, Valeria De Paiva, and Martin Hyland. Linear λ -calculus and categorical models revisited. In *International Workshop on Computer Science Logic*, pages 61–84. Springer, 1992.
- 7 Richard F Blute, J Robin B Cockett, and Robert AG Seely. Cartesian differential categories. *Theory and Applications of Categories*, 22(23):622–672, 2009.
- 8 Filippo Bonchi, Fabio Gadducci, Aleks Kissinger, Pawel Sobocinski, and Fabio Zanasi. Rewriting modulo symmetric monoidal structure. In Martin Grohe, Eric Koskinen, and Natarajan Shankar, editors, *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science, LICS '16, New York, NY, USA, July 5-8, 2016*, pages 710–719. ACM, 2016. doi:10.1145/2933575.2935316.
- 9 Filippo Bonchi, Pawel Sobocinski, and Fabio Zanasi. Deconstructing lawvere with distributive laws. *J. Log. Algebraic Methods Program.*, 95:128–146, 2018. doi:10.1016/j.jlamp.2017.12.002.
- 10 Alois Brunel, Damiano Mazza, and Michele Pagani. Backpropagation in the simply typed lambda-calculus with linear negation. *Proc. ACM Program. Lang.*, 4(POPL):64:1–64:27, 2020. doi:10.1145/3371132.
- 11 Albert Burroni. Higher-dimensional word problems with applications to equational logic. *Theoretical computer science*, 115(1):43–62, 1993.
- 12 Robin Cockett, Geoffrey Cruttwell, Jonathan Gallagher, Jean-Simon Pacaud Lemay, Benjamin MacAdam, Gordon Plotkin, and Dorette Pronk. Reverse derivative categories. *arXiv preprint*, 2019. arXiv:1910.07065.

- 13 Conal Elliott. The simple essence of automatic differentiation. *Proceedings of the ACM on Programming Languages*, 2(ICFP):1–29, 2018.
- 14 Dan R. Ghica. Geometry of synthesis: a structured approach to VLSI design. In Martin Hofmann and Matthias Felleisen, editors, *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17-19, 2007*, pages 363–375. ACM, 2007. doi:10.1145/1190216.1190269.
- 15 Jean-Yves Girard. Linear logic. *Theor. Comput. Sci.*, 50:1–102, 1987. doi:10.1016/0304-3975(87)90045-4.
- 16 Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and types*, volume 7. Cambridge university press Cambridge, 1989.
- 17 Stefano Guerrini. A general theory of sharing graphs. *Theor. Comput. Sci.*, 227(1-2):99–151, 1999. doi:10.1016/S0304-3975(99)00050-X.
- 18 Yves Guiraud. *Rewriting methods in higher algebra*. PhD thesis, Université Paris 7, 2019.
- 19 Chris Heunen and Jamie Vicary. Lectures on categorical quantum mechanics. *Computer Science Department. Oxford University*, 2012. URL: <http://www.cs.ox.ac.uk/files/4551/cqm-notes.pdf>.
- 20 Dominic Hughes. Simple free star-autonomous categories and full coherence. *CoRR*, 2005. arXiv:0506521.
- 21 Mathieu Huot, Sam Staton, and Matthijs Vákár. Correctness of automatic differentiation via diffeologies and categorical gluing. In Jean Goubault-Larrecq and Barbara König, editors, *Foundations of Software Science and Computation Structures - 23rd International Conference, FOSSACS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings*, volume 12077 of *Lecture Notes in Computer Science*, pages 319–338. Springer, 2020. doi:10.1007/978-3-030-45231-5_17.
- 22 Faustyna Krawiec, Simon Peyton Jones, Neel Krishnaswami, Tom Ellis, Richard A Eisenberg, and Andrew W Fitzgibbon. Provably correct, asymptotically efficient, higher-order reverse-mode automatic differentiation. *Proc. ACM Program. Lang.*, 6(POPL):1–30, 2022.
- 23 John Lamping. An algorithm for optimal lambda calculus reduction. In Frances E. Allen, editor, *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages, San Francisco, California, USA, January 1990*, pages 16–30. ACM Press, 1990. doi:10.1145/96709.96711.
- 24 Ian Mackie. Interaction nets for linear logic. *Theor. Comput. Sci.*, 247(1-2):83–140, 2000. doi:10.1016/S0304-3975(00)00198-5.
- 25 Paul-André Melliès. Functorial boxes in string diagrams. In Zoltán Ésik, editor, *Computer Science Logic, 20th International Workshop, CSL 2006, 15th Annual Conference of the EACSL, Szeged, Hungary, September 25-29, 2006, Proceedings*, volume 4207 of *Lecture Notes in Computer Science*, pages 1–30. Springer, 2006. doi:10.1007/11874683_1.
- 26 Koko Muroya and Dan R. Ghica. The dynamic geometry of interaction machine: A token-guided graph rewriter. *Log. Methods Comput. Sci.*, 15(4), 2019. doi:10.23638/LMCS-15(4:7)2019.
- 27 Barak A. Pearlmutter and Jeffrey Mark Siskind. Reverse-mode AD in a functional framework: Lambda the ultimate backpropagator. *ACM Trans. Program. Lang. Syst.*, 30(2):7:1–7:36, 2008. doi:10.1145/1330017.1330018.
- 28 Ralf Schweimeier and Alan Jeffrey. A categorical and graphical treatment of closure conversion. In Stephen D. Brookes, Achim Jung, Michael W. Mislove, and Andre Scedrov, editors, *Fifteenth Conference on Mathematical Foundations of Programming Semantics, MFPS 1999, Tulane University, New Orleans, LA, USA, April 28 - May 1, 1999*, volume 20 of *Electronic Notes in Theoretical Computer Science*, pages 481–511. Elsevier, 1999. doi:10.1016/S1571-0661(04)80090-2.
- 29 Peter Selinger. A survey of graphical languages for monoidal categories. In *New structures for physics*, pages 289–355. Springer, 2010. arXiv:0908.3347.

30 Michael Shulman. *-autonomous envelopes and 2-conservativity of duals. *CoRR*, 2020. [arXiv:2004.08487](https://arxiv.org/abs/2004.08487).

31 Jeffrey Mark Siskind and Barak A. Pearlmutter. Efficient implementation of a higher-order language with built-in AD. *CoRR*, abs/1611.03416, 2016. [arXiv:1611.03416](https://arxiv.org/abs/1611.03416).

32 Morten Heine Sørensen and Pawel Urzyczyn. *Lectures on the Curry-Howard isomorphism*. Elsevier, 2006.

33 Matthijs Vákár. Reverse AD at higher types: Pure, principled and denotationally correct. In Nobuko Yoshida, editor, *Programming Languages and Systems - 30th European Symposium on Programming, ESOP 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings*, volume 12648 of *Lecture Notes in Computer Science*, pages 607–634. Springer, 2021. doi:10.1007/978-3-030-72019-3_22.

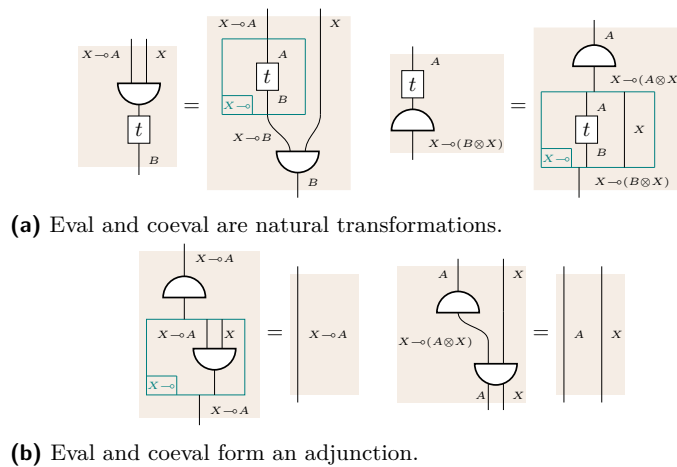
34 Jamie Vicary and Antonin Delpuch. Normalization for planar string diagrams and a quadratic equivalence algorithm. *Logical Methods in Computer Science*, 18, 2022.

35 Jamie Vicary, Aleks Kissinger, and Krzysztof Bar. Globular: an online proof assistant for higher-dimensional rewriting. *Logical Methods in Computer Science*, 14, 2018.

36 Fei Wang, Daniel Zheng, James M. Decker, Xilun Wu, Grégory M. Essertel, and Tiark Rompf. Demystifying differentiable programming: shift/reset the penultimate backpropagator. *Proc. ACM Program. Lang.*, 3(ICFP):96:1–96:31, 2019. doi:10.1145/3341700.

A Monoidal closed adjunctions, diagrammatically

Instantiated to the families of functors F_X, G_X of Section 2.3, the naturality and adjunction equations are expressed in string diagrams as in Fig. 13. The counit of the adjunction is normally called *eval*, and we call the unit *coeval* for the sake of symmetry in terminology and by analogy with compact-closed categories.



■ **Figure 13** String diagram representation of MCC axioms.

B Reverse differential categories

We present here the full definition of reverse differential categories. For a more exhaustive treatment of these, containing a discussion of the intuition behind the axioms and many useful properties of reverse differential categories, we refer to [12].

► **Definition 18** ([12, Def. 13]). *A reverse differential category is a cartesian left additive category endowed with a combinator R sending each morphism $f : X \rightarrow Y$ to a morphism $R[f] : X \times Y \rightarrow X$ which satisfies the following conditions:*

- (RD1) $R[0_X] = \omega_X$ and $R[+_X] = \omega_{X \times X} \times \delta_X$,
 - (RD2) $(id_X \times 0_Y); R[f] = \omega_X; 0_X$ and $(id_X \times +_Y); R[f] = (\delta_X \times id_{Y \times Y}); (id_X \times \gamma_{X,Y} \times id_Y); (R[f] \times R[f]); +_X$,
 - (RD3) $R[id_X] = \omega_X \times id_X$, $R[\omega_X] = \omega_X \times 0_X$, and $R[\delta_X] = \omega_X \times +_X$
 - (RD4) $R[f \times h] = (id_X \times \gamma_{Z,Y} \times id_W); (R[f] \times R[h])$,
 - (RD5) $R[f; g] = (\delta_X \times id_Z); (id_X \times f \times id_Z); (id_X \times R[g]); R[f]$
 - (RD6) $(id_{X \times Y} \times 0_{X \times X} \times id_Y); R^3[f]; (\omega_{X \times Y} \times id_X) = (id_X \times \omega_Y \times id_Y); R[f]$
 - (RD7) $(id_X \times 0_Y \times id_X \times 0_{X \times Y} \times id_X \times 0_Y \times id_X); R^4[f]; (\omega_{X \times Y \times X \times X} \times id_Y) = (id_X \times \gamma_{X,X} \times id_X); (id_X \times 0_Y \times id_X \times 0_{X \times Y} \times id_X \times 0_Y \times id_X); R^4[f]; (\omega_{X \times Y \times X \times X} \times id_Y)$
- for all $g : Y \rightarrow Z$ and $h : Z \rightarrow W$.

The careful reader will note that our formulation of these axioms differ from the formulation given by Cockett et al., but the two systems are equivalent and essentially result from our different formulation of left additive structure.

In words, RD1 and RD3 tell us the derivatives of the basic structural morphisms in a cartesian left additive category: RD1 gives the derivatives of the left additive structure, and RD3 gives the derivatives of the cartesian category. These are in some sense the base cases. Axioms RD4 and RD5 give induction steps, spelling out how the derivatives of morphisms composed in parallel (RD4) or in sequence (RD5) are constructed from the derivatives of its components. In particular, RD5 can be seen as a “reverse-mode chain rule.” The remaining axioms are not as important for our purposes, but RD2 states that the reverse derivative is additive in its small-change argument, RD6 broadly states that the reverse derivative of any map is linear in its second argument (in the sense that it coincides with its own derivative) and RD7 states that partial (reverse) derivatives commute.

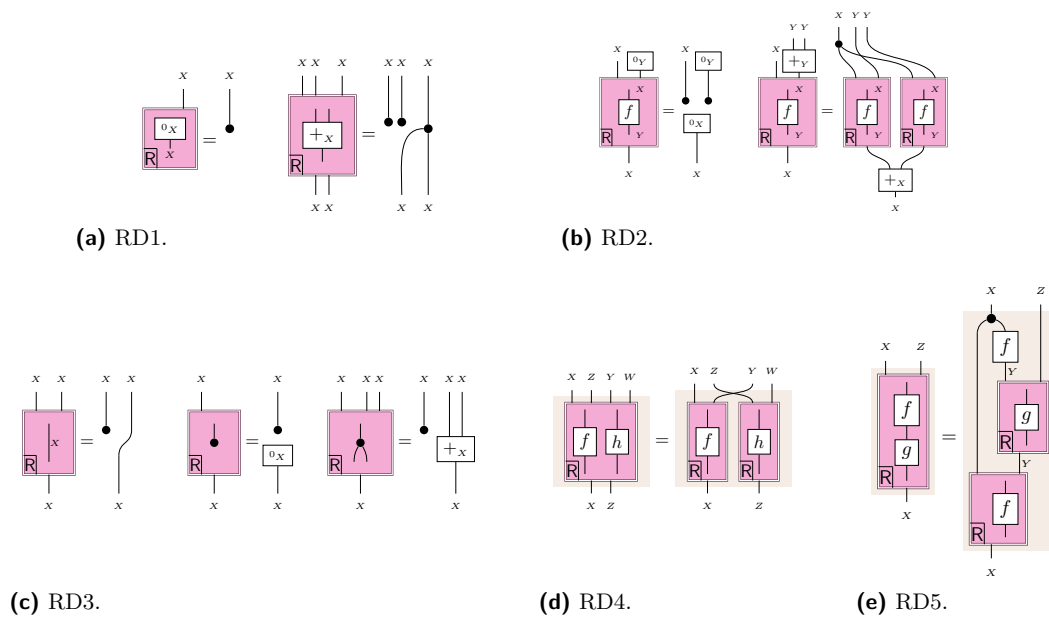
All of these axioms admit corresponding graphical presentations in terms of string diagrams, as can be found in Fig. 14. We only present here diagrams for the axioms that do not involve second derivatives, as they are the only ones relevant to our AD algorithm, but axioms RD6 and RD7 admit equally straightforward diagrammatic formulations.

► **Lemma 19.** *Suppose Σ is an operational signature and suppose that for every operation $g : X \rightarrow Y$ in Σ_1 , there is a well-defined reverse derivative $R[g]$ morphism in \mathcal{A}_Σ satisfying RD2, RD6, and RD7. Then \mathcal{A}_Σ is a reverse differential category.*

In such a case, we will call Σ a differentiable signature.

Proof. As noted previously, the RD axioms specify the derivatives of the structural morphisms from \mathcal{A}_Σ and the derivatives of composites. It is easy to check that RD2, RD6, and RD7 hold for the derivatives of these structural morphisms. Then this reduces to the (straightforward) check that when two composable morphisms satisfy RD2, RD6, and RD7 then their composite does as well. ◀

6:20 Functorial String Diagrams for RAD



■ Figure 14 RDC axioms as string diagrams.