


PAPER

CHAD for expressive total languages

Fernando Lucatelli Nunes and Matthijs Vákár 

Department of Information and Computing Sciences, Utrecht University, Utrecht, Netherlands

Corresponding author: Matthijs Vákár; Email: matthijsvakar@gmail.com

(Received 1 October 2021; revised 3 April 2023; accepted 22 May 2023; first published online 14 July 2023)

Abstract

We show how to apply forward and reverse mode Combinatory Homomorphic Automatic Differentiation (CHAD) (Vákár (2021). *ESOP*, 607–634; Vákár and Smeding (2022). *ACM Transactions on Programming Languages and Systems* 44 (3) 20:1–20:49.) to total functional programming languages with expressive type systems featuring the combination of

- tuple types;
- sum types;
- inductive types;
- coinductive types;
- function types.

We achieve this by analyzing the categorical semantics of such types in Σ -types (Grothendieck constructions) of suitable categories. Using a novel categorical logical relations technique for such expressive type systems, we give a correctness proof of CHAD in this setting by showing that it computes the usual mathematical derivative of the function that the original program implements. The result is a principled, purely functional and provably correct method for performing forward- and reverse-mode automatic differentiation (AD) on total functional programming languages with expressive type systems.

Keywords: Automatic differentiation; Software correctness; Programming languages; Scientific computing; Program transformations; Type systems; Dependently typed languages; Artin gluing; Comma categories; Logical relations; Initial algebra semantics; Creation of initial algebras; Coalgebras; Grothendieck construction; Exponentiability; Fibered categories; Polynomial functors; Linear types; Variant types; Inductive types; Coinductive types; Cartesian closed categories; Denotational semantics; Extensive indexed categories; Extensive categories; (Co)monadicity; Free cocompletion under coproducts

1. Introduction

Automatic differentiation (AD) is a popular technique for computing derivatives of functions implemented by computer programs, essentially by applying the chain rule across the program code. It is typically the method of choice for computing derivatives in machine learning and scientific computing because of its efficiency and numerical stability. AD has two main variants: forward-mode AD, which calculates the derivative of a function, and reverse-mode AD, which calculates the (matrix) transpose of the derivative. Roughly speaking, for a function $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$, reverse mode is the more efficient technique if $n \gg m$ and forward mode is if $n \ll m$. Seeing that we are usually interested in computing derivatives (or gradients) of functions $f : \mathbb{R}^n \rightarrow \mathbb{R}$ with very large n , reverse AD tends to be the more important algorithm in practice (Baydin et al. 2017).

While the study of AD has a long history in the numerical methods community, which we will not survey (see, e.g., Griewank and Walther 2008), there has recently been a proliferation of work by the programming languages community examining the technique from a new angle. New goals pursued by this community include

- giving a *concise, clear, and easy-to-implement definition* of various AD algorithms;
- *expanding the languages and programming techniques* that AD can be applied to;
- relating AD to its mathematical *foundations* in differential geometry and proving that AD implementations *correctly* calculate derivatives;
- performing AD at *compile time* through *source code transformation*, to maximally expose optimization opportunities to the compiler and to avoid interpreter overhead that other AD approaches can incur;
- providing formal *complexity guarantees* for AD implementations.

We provide a brief summary of some of this more recent work in Section 16. The present paper adds to this new body of work by advancing the state of the art of the first four goals. We leave the fifth goal when applied to our technique mostly to future work (with the exception of Corollary 130). Specifically, we extend the scope of the Combinatory Homomorphic Automatic Differentiation (CHAD) method of forward and reverse AD (Vákár 2021; Vákár and Smeding 2022) (from the previous state of the art: a simply typed λ -calculus) to apply to total functional programming languages with expressive type systems, that is, the combination of:

- *tuple types*, to enable programs that return or take as an argument more than one value;
- *sum types*, to enable programs that define and branch on variant data types;
- *inductive types*, to include programs that operate on labeled-tree-like data structures;
- *coinductive types*, to deal with programs that operate on lazy infinite data structures such as streams;
- *function types*, to encompass programs that use popular higher-order programming idioms such as maps and folds.

This conceptually simple extension requires a considerable extension of existing techniques in denotational semantics. The payoffs of this challenging development are surprisingly simple AD algorithms as well as reusable abstract semantic techniques.

The main contributions of this paper are as follows:

- developing an abstract categorical semantics (Section 3) of such expressive type systems in suitable Σ -types of categories (Section 6);
- presenting, as the initial instantiation of this abstract semantics, an idealized target language for CHAD when applied to such type systems (Section 7);
- deriving the forward and reverse CHAD algorithms (Section 8) when applied to expressive type systems as the uniquely defined homomorphic functors (Section 4) from the source (Section 5) to the target language (Section 7);
- introducing (categorical) logical relations techniques (aka scoping) for reasoning about expressive functional languages that include both inductive and coinductive types (Section 11);
- using such a logical relations construction over the concrete denotational semantics (Section 10) of the source and target languages (Section 9) that demonstrates that CHAD correctly calculates the usual mathematical derivative (Section 12), even for programs between inductive types (Section 13);
- discussing examples (Section 14) and applied considerations around implementing this extended CHAD method in practice (Section 15).

We start by giving a high-level overview of the key insights and theorems in this paper in Section 2.

2. Key Ideas

2.1 Origins in semantic derivatives and chain rules

CHAD starts from the observation that for a differentiable function:

$$f : \mathbb{R}^n \rightarrow \mathbb{R}^m$$

it is useful to pair the primal function value $f(x)$ with f 's derivative $Df(x)$ at x if we want to calculate derivatives in a compositional way (where we underline the spaces \mathbb{R}^n of tangent vectors to emphasize their algebraic structure and we write a linear function type for the derivative to indicate its linearity in its tangent vector argument):

$$\begin{aligned} \mathcal{T}f : \mathbb{R}^n &\rightarrow \mathbb{R}^m \times (\underline{\mathbb{R}^n} \multimap \underline{\mathbb{R}^m}) \\ x &\mapsto (f(x), Df(x)). \end{aligned}$$

Indeed, the chain rule for derivatives teaches us that we compute the derivative of a composition $g \circ f$ of functions as follows, where we write $\mathcal{T}_1f \stackrel{\text{def}}{=} \pi_1 \circ \mathcal{T}f$ and $\mathcal{T}_2f \stackrel{\text{def}}{=} \pi_2 \circ \mathcal{T}f$ for the first and second components of $\mathcal{T}f$, respectively:

$$\mathcal{T}(g \circ f)(x) = (\mathcal{T}_1g(\mathcal{T}_1f(x)), \mathcal{T}_2g(\mathcal{T}_1f(x)) \circ \mathcal{T}_2f(x)).$$

We make two observations:

- (1) the derivative of $g \circ f$ does depend not only on the derivatives of g and f but also on the primal value of f ;
- (2) the primal value of f is used twice: once in the primal value of $g \circ f$ and once in its derivative; we want to share these repeated subcomputations.

Insight 1. This shows that it is wise to *pair up* computations of primal function values and derivatives and to *share* computation between both if we want to calculate derivatives of functions compositionally and efficiently.

Similar observations can be made for f 's transposed (adjoint) derivative Df^t , which propagates not tangent vectors but cotangent vectors and which we can pair up as:

$$\begin{aligned} \mathcal{T}^*f : \mathbb{R}^n &\rightarrow \mathbb{R}^m \times (\underline{\mathbb{R}^m} \multimap \underline{\mathbb{R}^n}) \\ x &\mapsto (f(x), Df^t(x)) \end{aligned}$$

to get the following chain rule:

$$\mathcal{T}^*(g \circ f)(x) = (\mathcal{T}_1^*g(\mathcal{T}_1^*f(x)), \mathcal{T}_2^*f(x) \circ \mathcal{T}_2^*g(\mathcal{T}_1^*f(x))).$$

CHAD directly implements the operations \mathcal{T} and \mathcal{T}^* as source code transformations $\overrightarrow{\mathcal{D}}$ and $\overleftarrow{\mathcal{D}}$ on a functional language to implement forward- and reverse-mode AD, respectively. These code transformations are defined compositionally through structural induction on the syntax, by making use of the chain rules above.

2.2 CHAD on a first-order functional language

We first discuss what the technique looks like on a standard typed first-order functional language. Despite our different presentation in terms of a λ -calculus rather than Elliott's categorical combinators, this is essentially the algorithm of Elliott (2018). Types τ, σ, ρ are either statically sized

arrays of n real numbers \mathbf{real}^n or tuples $\tau * \sigma$ of types τ, σ . We consider programs t of type σ in typing context $\Gamma = x_1 : \tau_1, \dots, x_n : \tau_n$, where x_i are identifiers. We write such a typing judgment for programs in context as $\Gamma \vdash t : \sigma$. As long as our language has certain primitive operations (which we represent schematically)

$$\frac{\Gamma \vdash t_1 : \mathbf{real}^{n_1} \quad \dots \quad \Gamma \vdash t_k : \mathbf{real}^{n_k}}{\Gamma \vdash \text{op}(t_1, \dots, t_k) : \mathbf{real}^m}$$

such as constants (as nullary operations), (elementwise) addition and multiplication of arrays, inner products and certain nonlinear functions like sigmoid functions, we can write complex programs by sequencing together such operations. For example, writing \mathbf{real} for \mathbf{real}^1 , we can write a program $x_1 : \mathbf{real}, x_2 : \mathbf{real}, x_3 : \mathbf{real}, x_4 : \mathbf{real} \vdash s : \mathbf{real}$ by:

```

let  $y = x_1 * x_4 + 2 * x_2$  in
let  $z = y * x_3$  in
let  $w = z + x_4$  in  $\sin(w)$ ,
    
```

where we indicate shared subcomputations with **let**-bindings.

CHAD observes that we can define for each language type τ associated types of

- forward-mode primal values $\vec{\mathcal{D}}(\tau)_1$;
we define $\vec{\mathcal{D}}(\mathbf{real}^n) = \mathbf{real}^n$ and $\vec{\mathcal{D}}(\tau * \sigma)_1 = \vec{\mathcal{D}}(\tau)_1 * \vec{\mathcal{D}}(\sigma)_1$, that is, for now $\vec{\mathcal{D}}(\tau)_1 = \tau$;
- reverse-mode primal values $\overleftarrow{\mathcal{D}}(\tau)_1$;
we define $\overleftarrow{\mathcal{D}}(\mathbf{real}^n) = \mathbf{real}^n$ and $\overleftarrow{\mathcal{D}}(\tau * \sigma)_1 = \overleftarrow{\mathcal{D}}(\tau)_1 * \overleftarrow{\mathcal{D}}(\sigma)_1$; that is, for now $\overleftarrow{\mathcal{D}}(\tau)_1 = \tau$;
- forward-mode tangent values $\vec{\mathcal{D}}(\tau)_2$;
we define $\vec{\mathcal{D}}(\mathbf{real}^n)_2 = \mathbf{real}^n$ and $\vec{\mathcal{D}}(\tau * \sigma)_2 = \vec{\mathcal{D}}(\tau)_2 * \vec{\mathcal{D}}(\sigma)_2$;
- reverse-mode cotangent values $\overleftarrow{\mathcal{D}}(\tau)_2$;
we define $\overleftarrow{\mathcal{D}}(\mathbf{real}^n)_2 = \mathbf{real}^n$ and $\overleftarrow{\mathcal{D}}(\tau * \sigma)_2 = \overleftarrow{\mathcal{D}}(\tau)_2 * \overleftarrow{\mathcal{D}}(\sigma)_2$.

Indeed, the justification for these definitions is the crucial observation that a (co)tangent vector to a product of spaces is precisely a pair of tangent (co)vectors to the two spaces. Put differently, the space $\mathcal{T}_{(x,y)}(X \times Y)$ of (co)tangent vectors to $X \times Y$ at a point (x, y) equals the product space $(\mathcal{T}_x X) \times (\mathcal{T}_y Y)$ (Tu 2011).

We write the (co)tangent types associated with \mathbf{real}^n as $\underline{\mathbf{real}}^n$ to emphasize that it is a linear type and to distinguish it from the cartesian type \mathbf{real}^n . In particular, we will see that tangent and cotangent values are elements of linear types that come equipped with a commutative monoid structure $(\underline{0}, +)$. Indeed, (transposed) derivatives are linear functions: homomorphisms of this monoid structure¹. We extend these operations $\vec{\mathcal{D}}$ and $\overleftarrow{\mathcal{D}}$ to act on typing contexts Γ :

$$\begin{aligned} \vec{\mathcal{D}}(x_1 : \tau_1, \dots, x_n : \tau_n)_1 &= x_1 : \vec{\mathcal{D}}(\tau_1)_1, \dots, x_n : \vec{\mathcal{D}}(\tau_n)_1 \\ \overleftarrow{\mathcal{D}}(x_1 : \tau_1, \dots, x_n : \tau_n)_1 &= x_1 : \overleftarrow{\mathcal{D}}(\tau_1)_1, \dots, x_n : \overleftarrow{\mathcal{D}}(\tau_n)_1 \\ \vec{\mathcal{D}}(x_1 : \tau_1, \dots, x_n : \tau_n)_2 &= \vec{\mathcal{D}}(\tau_1)_2 * \dots * \vec{\mathcal{D}}(\tau_n)_2 \\ \overleftarrow{\mathcal{D}}(x_1 : \tau_1, \dots, x_n : \tau_n)_2 &= \overleftarrow{\mathcal{D}}(\tau_1)_2 * \dots * \overleftarrow{\mathcal{D}}(\tau_n)_2. \end{aligned}$$

To each program $\Gamma \vdash t : \sigma$, CHAD associates programs calculating the forward-mode and reverse-mode derivatives $\vec{\mathcal{D}}_{\overline{\Gamma}}(t)$ and $\overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(t)$, which are indexed by the list $\overline{\Gamma}$ of identifiers that occur in Γ :

$$\vec{\mathcal{D}}(\overline{\Gamma})_1 \vdash \overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(t) : \vec{\mathcal{D}}(\sigma)_1 * \left(\vec{\mathcal{D}}(\overline{\Gamma})_2 \multimap \vec{\mathcal{D}}(\sigma)_2 \right)$$

$$\overleftarrow{\mathcal{D}}(\Gamma)_1 \vdash \overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(t) : \overleftarrow{\mathcal{D}}(\sigma) * \left(\overleftarrow{\mathcal{D}}(\sigma) \multimap \overleftarrow{\mathcal{D}}(\Gamma)_2 \right).$$

Observing that each program t computes a differentiable function $\llbracket t \rrbracket$ between Euclidean spaces, as long as all primitive operations op are differentiable, the key property that we prove for these code transformations is that they actually calculate derivatives:

Theorem A (Correctness of CHAD, Theorem 124). *For any well-typed program:*

$$x_1 : \mathbf{real}^{m_1}, \dots, x_k : \mathbf{real}^{m_k} \vdash t : \mathbf{real}^m$$

we have that $\llbracket \overrightarrow{\mathcal{D}}_{x_1, \dots, x_k}(t) \rrbracket = \mathcal{T}_{\llbracket t \rrbracket}$ and $\llbracket \overleftarrow{\mathcal{D}}_{x_1, \dots, x_k}(t) \rrbracket = \mathcal{T}^* \llbracket t \rrbracket$.

Once we fix the semantics for the source and target languages, we can show that this theorem holds if we define $\overrightarrow{\mathcal{D}}$ and $\overleftarrow{\mathcal{D}}$ on programs using the chain rule. The proof works by plain induction on the syntax. For example, we can correctly define reverse-mode CHAD on a first-order language as follows:

$$\begin{aligned} \overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(\text{op}(t_1, \dots, t_k)) \stackrel{\text{def}}{=} & \quad \mathbf{let} \langle x_1, x'_1 \rangle = \overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(t_1) \mathbf{in} \dots \\ & \quad \mathbf{let} \langle x_k, x'_k \rangle = \overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(t_k) \mathbf{in} \\ & \quad \langle \text{op}(x_1, \dots, x_k), \underline{\lambda}v. \mathbf{let} v = \text{Dop}^t(x_1, \dots, x_k; v) \mathbf{in} \\ & \quad \quad x'_1 \bullet \mathbf{proj}_1 v + \dots + x'_k \bullet \mathbf{proj}_k v \rangle \end{aligned}$$

$$\overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(x) \stackrel{\text{def}}{=} \langle x, \underline{\lambda}v. \mathbf{coproj}_{\text{idx}(x; \overline{\Gamma})}(v) \rangle$$

$$\begin{aligned} \overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(\mathbf{let} x = t \mathbf{in} s) \stackrel{\text{def}}{=} & \quad \mathbf{let} \langle x, x' \rangle = \overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(t) \mathbf{in} \\ & \quad \mathbf{let} \langle y, y' \rangle = \overleftarrow{\mathcal{D}}_{\overline{\Gamma}, x}(s) \mathbf{in} \\ & \quad \langle y, \underline{\lambda}v. \mathbf{let} v = y' \bullet v \mathbf{in} \mathbf{fst} v + x' \bullet (\mathbf{snd} v) \rangle \end{aligned}$$

$$\begin{aligned} \overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(\langle t, s \rangle) \stackrel{\text{def}}{=} & \quad \mathbf{let} \langle x, x' \rangle = \overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(t) \mathbf{in} \\ & \quad \mathbf{let} \langle y, y' \rangle = \overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(s) \mathbf{in} \\ & \quad \langle \langle x, y \rangle, \underline{\lambda}v. x' \bullet (\mathbf{fst} v) + y' \bullet (\mathbf{snd} v) \rangle \end{aligned}$$

$$\overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(\mathbf{fst} t) \stackrel{\text{def}}{=} \mathbf{let} \langle x, x' \rangle = \overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(t) \mathbf{in} \langle \mathbf{fst} x, \underline{\lambda}v. x' \bullet \langle v, \underline{0} \rangle \rangle$$

$$\overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(\mathbf{snd} t) \stackrel{\text{def}}{=} \mathbf{let} \langle x, x' \rangle = \overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(t) \mathbf{in} \langle \mathbf{snd} x, \underline{\lambda}v. x' \bullet \langle \underline{0}, v \rangle \rangle$$

Here, we write $\underline{\lambda}v.t$ for a linear function abstraction (merely a notational convention – it can simply be thought of as a plain function abstraction) and $t \bullet s$ for a linear function application (which again can be thought of as a plain function application). Furthermore, given $\Gamma; v : \underline{\alpha} \vdash t : (\underline{\sigma}_1 * \dots * \underline{\sigma}_n)$, we write $\Gamma; v : \underline{\alpha} \vdash \mathbf{proj}_i(t) : \underline{\sigma}_i$ for the i -th projection of t . Similarly, given $\Gamma; v : \underline{\alpha} \vdash t : \underline{\sigma}_i$, we write the i -th coprojection $\Gamma; v : \underline{\alpha} \vdash \mathbf{coproj}_i(t) = \langle \underline{0}, \dots, \underline{0}, t, \underline{0}, \dots, \underline{0} \rangle : (\underline{\sigma}_1 * \dots * \underline{\sigma}_n)$ and we write $\text{idx}(x_i; x_1, \dots, x_n) = i$ for the index of an identifier in a list of identifiers. Finally, Dop^t here is a linear operation that implements the transposed derivative of the primitive operation op .

Note, in particular, that CHAD pairs up primal and (co)tangent values and shares common subcomputations. We see that what CHAD achieves is a compositional efficient reverse-mode AD algorithm that computes the (transposed) derivatives of a composite program in terms of the (transposed) derivatives Dop^t of the basic building blocks op .

2.3 CHAD on a higher-order language: a categorical perspective saves the day

So far, this account of CHAD has been smooth sailing: we can simply follow the usual mathematics of (transposed) derivatives of functions $\mathbb{R}^n \rightarrow \mathbb{R}^m$ and implement it in code. A challenge arises when trying to extend the algorithm to more expressive languages with features that do not have an obvious counterpart in multivariate calculus, like higher-order functions.

Vákár and Smeding (2022) and Vákár (2021) solve this problem by observing that we can understand CHAD through the categorical structure of Grothendieck constructions (aka Σ -types of categories). In particular, they observe that the syntactic category of the target language for CHAD, a language with both cartesian and linear types, forms a locally indexed category $\mathbf{LSyn} : \mathbf{CSyn}^{op} \rightarrow \mathbf{Cat}$, that is, functor to the category of categories and functors for which $\text{obj}(\mathbf{LSyn})(\tau) = \text{obj}(\mathbf{LSyn})(\sigma)$ for all $\tau, \sigma \in \text{obj}(\mathbf{CSyn})$ and $\mathbf{LSyn}(\tau \xrightarrow{t} \sigma) : \mathbf{LSyn}(\sigma) \rightarrow \mathbf{LSyn}(\tau)$ is identity on objects. Here, \mathbf{CSyn} is the syntactic category whose objects are cartesian types τ, σ, ρ and morphisms $\tau \rightarrow \sigma$ are programs $x : \tau \vdash t : \sigma$, up to a standard program equivalence. Similarly, $\mathbf{LSyn}(\tau)$ is the syntactic category whose objects are linear types $\underline{\alpha}, \underline{\sigma}, \underline{\gamma}$ and morphisms $\underline{\alpha} \rightarrow \underline{\gamma}$ are programs $x : \tau; v : \underline{\alpha} \vdash t : \underline{\gamma}$ of type $\underline{\gamma}$ that have a free variable x of cartesian type τ and a free variable v of linear type $\underline{\alpha}$. The key observation then is the following.

Theorem B (CHAD from a universal property, Corollary 69). *Forward- and reverse-mode CHAD are the unique structure-preserving functors:*

$$\begin{aligned} \overrightarrow{D}(-) : \mathbf{Syn} &\rightarrow \Sigma_{\mathbf{CSyn}} \mathbf{LSyn} \\ \overleftarrow{D}(-) : \mathbf{Syn} &\rightarrow \Sigma_{\mathbf{CSyn}} \mathbf{LSyn}^{op} \end{aligned}$$

from the syntactic category \mathbf{Syn} of the source language to (opposite) Grothendieck construction of the target language $\mathbf{LSyn} : \mathbf{CSyn}^{op} \rightarrow \mathbf{Cat}$ that send primitive operations op to their derivative Dop and transposed derivative Dop^t , respectively.

In particular, they prove that this is true for the unambiguous definitions of CHAD for a source language that is the first-order functional language we have considered above, which we can see as the freely generated category \mathbf{Syn} with finite products, generated by the objects \mathbf{real}^n and morphisms op . That is, for this limited language, “structure-preserving functor” should be interpreted as “finite product-preserving functor.”

This leads (Vákár 2021; Vákár and Smeding 2022) to the idea to try to use Theorem B as a definition of CHAD on more expressive programming languages. In particular, they consider a higher-order functional source language \mathbf{Syn} , that is, the freely generated cartesian closed category on the objects \mathbf{real}^n and morphisms op and try to define $\overrightarrow{D}(-)$ and $\overleftarrow{D}(-)$ as the (unique) structure-preserving (meaning: cartesian closed) functors to $\Sigma_{\mathbf{CSyn}} \mathbf{LSyn}$ and $\Sigma_{\mathbf{CSyn}} \mathbf{LSyn}^{op}$ for a suitable linear target language $\mathbf{LSyn} : \mathbf{CSyn}^{op} \rightarrow \mathbf{Cat}$. The main contribution then is to identify conditions on a locally indexed category $\mathcal{L} : \mathcal{C}^{op} \rightarrow \mathbf{Cat}$ that guarantee that $\Sigma_{\mathcal{L}} \mathcal{L}$ and $\Sigma_{\mathcal{L}} \mathcal{L}^{op}$ are cartesian closed and to take the target language $\mathbf{LSyn} : \mathbf{CSyn}^{op} \rightarrow \mathbf{Cat}$ as a freely generated such category.

Insight 2. To understand how to perform CHAD on a source language with language feature X (e.g., higher-order functions), we need to understand the categorical semantics of language feature X (e.g., categorical exponentials) in categories of the form $\Sigma_{\mathcal{L}} \mathcal{L}$ and $\Sigma_{\mathcal{L}} \mathcal{L}^{op}$. Giving sufficient conditions on \mathcal{L} for such a semantics to exist yields a suitable target language for CHAD, with the definition of the algorithm falling from the universal property of the source language.

Furthermore, we observe in these papers that Theorem A again holds for this extended definition of CHAD on higher-order languages. However, to prove this, plain induction no longer suffices and we instead need to use a logical relations construction over the semantics (in the form of categorical scoping) that relates differentiable curves to their associated primal and (co)tangent

curves. This is necessary because the program t may use higher-order constructions such as λ -abstractions and function applications in its definition, even if the input and output types are plain first-order types that implement some Euclidean space.

Insight 3. To obtain a correctness proof of CHAD on source languages with language feature X , it suffices to give a concrete denotational semantics for the source and target languages as well as a categorical semantics of language feature X in a category of logical relations (a scone) over these concrete semantics. The main technical challenge is to analyze logical relations techniques for language feature X .

Finally, these papers observe that the resulting target language can be implemented as a shallowly embedded DSL in standard functional languages, using a module system to implement the required linear types as abstract types, with a reference Haskell implementation available at <https://github.com/VMatthijs/CHAD>. In fact, Vytiniotis et al. (2019) had proposed the same CHAD algorithm for higher-order languages, arriving at it from practical considerations rather than abstract categorical observations.

Insight 4. The code generated by CHAD naturally comes equipped with very precise (e.g., linear) types. These types emphasize the connections to its mathematical foundations and provide scaffolding for its correctness proof. However, they are unnecessary for a practical implementation of the algorithm: CHAD can be made to generate standard functional (e.g., Haskell) code; the type safety can even be rescued by implementing the linear types as abstract types.

2.4 CHAD for sum types: a challenge – (co)tangent spaces of varying dimension

A natural approach, therefore, when extending CHAD to yet more expressive source languages is to try to use Theorem B as a definition. In the case of sum types (aka variant types), therefore, we should consider their categorical equivalent, distributive coproducts, and seek conditions on $\mathcal{L} : \mathcal{C}^{op} \rightarrow \mathbf{Cat}$ under which $\Sigma_{\mathcal{C}}\mathcal{L}$ and $\Sigma_{\mathcal{C}}\mathcal{L}^{op}$ have distributive coproducts. The difficulty is that these categories tend not to have coproducts if \mathcal{L} is locally indexed. Instead, the desire to have coproducts in $\Sigma_{\mathcal{C}}\mathcal{L}$ and $\Sigma_{\mathcal{C}}\mathcal{L}^{op}$ naturally leads us to consider more general strictly indexed categories $\mathcal{L} : \mathcal{C}^{op} \rightarrow \mathbf{Cat}$.

In fact, this is compatible with what we know from differential geometry (Tu 2011): coproducts allow us to construct spaces with multiple connected components, each of which may have a distinct dimension. To make things concrete, the space $\mathcal{T}_x(\mathbb{R}^2 \sqcup \mathbb{R}^3)$ of tangent vectors to $\mathbb{R}^2 \sqcup \mathbb{R}^3$ is either \mathbb{R}^2 or \mathbb{R}^3 depending on whether the base point x is chosen in the left or right component of the coproduct. More generally, a differentiable function $f : X \rightarrow Y$ between spaces of varying dimension (which can be formalized as manifolds with multiple connected components) induces functions on the spaces of tangent and cotangent vectors²:

$$\begin{aligned} \mathcal{T}f &: \prod_{x \in X} \Sigma_{y \in Y} (\mathcal{T}_x X \multimap \mathcal{T}_y Y) \\ \mathcal{T}^*f &: \prod_{x \in X} \Sigma_{y \in Y} (\mathcal{T}_y^* Y \multimap \mathcal{T}_x^* X), \end{aligned}$$

whose first component is f itself and whose second component is the action on (co)tangent vectors that f induces.

If the types $\overrightarrow{\mathcal{D}}(\tau)_2$ and $\overleftarrow{\mathcal{D}}(\tau)_2$ are to represent spaces of tangent and cotangent vectors to the spaces that $\overrightarrow{\mathcal{D}}(\tau)_1$ and $\overleftarrow{\mathcal{D}}(\tau)_1$ represent, we would expect them to be types that vary with the particular base point (primal) we choose. This leads to a refined view of CHAD: while $\vdash \overrightarrow{\mathcal{D}}(\tau)_1 : \text{type}$ and $\vdash \overleftarrow{\mathcal{D}}(\tau)_1 : \text{type}$ can remain (closed/nondependent) cartesian types, $p : \overrightarrow{\mathcal{D}}(\tau)_1 \vdash \overrightarrow{\mathcal{D}}(\tau)_2 : \text{type}$ and $p : \overleftarrow{\mathcal{D}}(\tau)_1 \vdash \overleftarrow{\mathcal{D}}(\tau)_2 : \text{type}$ are, in general, linear dependent types.

Insight 5. To accommodate sum types in CHAD, it is natural to consider a target language with dependent types: this allows the dimension of the spaces of (co)tangent vectors to vary with the chosen primal. In categorical terms, we need to consider general strictly indexed categories $\mathcal{L} : \mathcal{C}^{op} \rightarrow \mathbf{Cat}$ instead of merely locally indexed ones.

The CHAD transformations of the program now becomes typed in the following more precise way:

$$\begin{aligned} \vec{D}(\Gamma)_1 \vdash \vec{D}_{\Gamma}(t) : \Sigma p : \vec{D}(\tau)_1. \vec{D}(\Gamma)_2 \multimap \vec{D}(\tau)_2 \\ \overleftarrow{D}(\Gamma)_1 \vdash \overleftarrow{D}_{\Gamma}(t) : \Sigma p : \overleftarrow{D}(\tau)_1. \overleftarrow{D}(\Gamma)_2 \multimap \overleftarrow{D}(\tau)_2, \end{aligned}$$

where the action of $\vec{D}(-)_2$ and $\overleftarrow{D}(-)_2$ on typing contexts $\Gamma = x_1 : \tau_1, \dots, x_n : \tau_n$ has been refined to

$$\vec{D}(\Gamma)_2 \stackrel{\text{def}}{=} (\vec{D} \tau_1)_2[x_1/p] * \dots * (\vec{D} \tau_n)_2[x_n/p] \quad \overleftarrow{D}(\Gamma)_2 \stackrel{\text{def}}{=} (\overleftarrow{D}(\tau_1)_2[x_1/p]) * \dots * (\overleftarrow{D}(\tau_n)_2[x_n/p]).$$

All given definitions remain valid, where we simply reinterpret some tuples as having a Σ -type rather than the more limited original tuple type.

We prove the following novel results.

Theorem C (Bicartesian closed structure of Σ -categories, Propositions 17 and 18, Theorems 25, 26, and 39, and Corollaries 35 and 36). *For a category \mathcal{C} and a strictly indexed category $\mathcal{L} : \mathcal{C}^{op} \rightarrow \mathbf{Cat}$, $\Sigma_{\mathcal{C}}\mathcal{L}$ and $\Sigma_{\mathcal{C}}\mathcal{L}^{op}$ have*

- (fibered) finite products, if \mathcal{C} has finite coproducts and \mathcal{L} has strictly indexed products and coproducts;
- (fibered) finite coproducts, if \mathcal{C} has finite coproducts and \mathcal{L} is extensive;
- exponentials, if \mathcal{L} is a biadditive model of the dependently typed enriched effect calculus (we intentionally keep this vague here to aid legibility – the point is that these are relatively standard conditions).

Furthermore, the coproducts in $\Sigma_{\mathcal{C}}\mathcal{L}$ and $\Sigma_{\mathcal{C}}\mathcal{L}^{op}$ distribute over the products, as long as those in \mathcal{C} do, even in the absence of exponentials. Notably, the exponentials are not generally fibered over \mathcal{C} .

The crucial notion here is our (novel) notion of extensivity of an indexed category, which generalizes well-known notions of extensive categories. In particular, we call $\mathcal{L} : \mathcal{C}^{op} \rightarrow \mathbf{Cat}$ extensive if the canonical functor $\mathcal{L}(\sqcup_{i=1}^n C_i) \rightarrow \prod_{i=1}^n \mathcal{L}(C_i)$ is an equivalence. Furthermore, we note that we need to reestablish the product and exponential structures of $\Sigma_{\mathcal{C}}\mathcal{L}$ and $\Sigma_{\mathcal{C}}\mathcal{L}^{op}$ due to the generalization from locally indexed to arbitrary strictly indexed categories \mathcal{L} .

Using these results, we construct a suitable target language **LSyn** : **CSyn**^{op} \rightarrow **Cat** for CHAD on a source language with sum types (and tuple and function types) and derive the forward and reverse CHAD algorithms for such a language and reestablish Theorems A and B in this more general context. This target language is a standard dependently typed enriched effect calculus with cartesian sum types and extensive families of linear types (i.e., dependent linear types that can be defined through case distinction). Again, the correctness proof of Theorem A uses the universal property of Theorem B and a logical relations (categorical scoping) construction over the denotational semantics of the source and target languages. This logical relations construction is relatively straightforward and relies on well-known scoping methods for bicartesian closed categories. In particular, we obtain the following formulas for a sum type $\{\ell_1 \tau_1 \mid \dots \mid \ell_n \tau_n\}$ with constructors ℓ_1, \dots, ℓ_n that take arguments of type τ_1, \dots, τ_n :

$$\begin{aligned} \vec{D} \{\ell_1 \tau_1 \mid \dots \mid \ell_n \tau_n\}_1 &\stackrel{\text{def}}{=} \{ \ell_1 \vec{D} \tau_1 \mid \dots \mid \ell_n \vec{D} \tau_n \} \\ \vec{D} \{\ell_1 \tau_1 \mid \dots \mid \ell_n \tau_n\}_2 &\stackrel{\text{def}}{=} \mathbf{case } p \mathbf{ of } \{ \ell_1 p \rightarrow \vec{D} \tau_1 \mid \dots \mid \ell_n p \rightarrow \vec{D} \tau_n \} \end{aligned}$$

$$\begin{aligned} \overleftarrow{\mathcal{D}}(\{\ell_1\tau_1 \mid \dots \mid \ell_n\tau_n\})_1 &\stackrel{\text{def}}{=} \left\{ \ell_1 \overleftarrow{\mathcal{D}}(\tau_1)_1 \mid \dots \mid \ell_n \overleftarrow{\mathcal{D}}(\tau_n)_1 \right\} \\ \overleftarrow{\mathcal{D}}(\{\ell_1\tau_1 \mid \dots \mid \ell_n\tau_n\})_2 &\stackrel{\text{def}}{=} \text{case } p \text{ of } \{\ell_1 p \rightarrow \overrightarrow{\mathcal{D}}\tau_1\}_2 \mid \dots \mid \ell_n p \rightarrow \overleftarrow{\mathcal{D}}(\tau_n)_2, \end{aligned}$$

mirroring our intuition that the (co)tangent bundle to a coproduct of spaces decomposes (extensively) into the (co)tangent bundles to the component spaces.

2.5 CHAD for (co)inductive types: where do we begin?

If we are to really push forward the dream of differentiable programming, we need to learn how to perform AD on programs that operate on data types. To this effect, we analyze CHAD for inductive and coinductive types. If we want to follow our previous methodology to find suitable definitions and correctness proofs, we first need a good categorical axiomatization of such types. It is well known that inductive types correspond to initial algebras of functors, while coinductive types are precisely terminal coalgebras. The question, however, is what class of functors to consider. That choice makes the vague notion of (co)inductive types precise.

Following Santocanale (2002), we work with the class of $\mu\nu$ -polynomials, a relatively standard choice, that is functors that can be defined inductively through the combination of

- constants for primitive types \mathbf{real}^n ;
- type variables α ;
- unit and tuple types $\mathbf{1}$ and $\tau * \sigma$ of $\mu\nu$ -polynomials;
- sum types $\{\ell_1\tau_1 \mid \dots \mid \ell_n\tau_n\}$ of $\mu\nu$ -polynomials;
- initial algebras $\mu\alpha.\tau$ of $\mu\nu$ -polynomials;
- terminal coalgebras $\nu\alpha.\tau$ of $\mu\nu$ -polynomials.

Notably, we exclude function types, as the non-fibered nature of exponentials in $\Sigma_C\mathcal{L}$ and $\Sigma_C\mathcal{L}^{op}$ would significantly complicate the technical development. While this excludes certain examples like the free state monad (which for type σ state would be the initial algebra $\mu\alpha.\{Get(\sigma \rightarrow \alpha) \mid Put(\sigma * \alpha)\}$), it still includes the vast majority of examples of eager and lazy types that one uses in practice, for example, lists $\mu\alpha.\{Empty \mathbf{1} \mid Cons(\sigma * \alpha)\}$, (finitely branching) labeled trees like $\mu\alpha.\{Leaf \mathbf{1} \mid Node(\sigma * \alpha * \alpha)\}$, streams $\nu\alpha.\sigma * \alpha$, and many more.

We characterize conditions on a strictly indexed category $\mathcal{L} : \mathcal{C}^{op} \rightarrow \mathbf{Cat}$ that guarantee that $\Sigma_C\mathcal{L}$ and $\Sigma_C\mathcal{L}^{op}$ have this precise notion of inductive and coinductive types. The first step is to give a characterization of initial algebras and terminal coalgebras of split fibration endofunctors on $\Sigma_C\mathcal{L}$ and $\Sigma_C\mathcal{L}^{op}$. For legibility, we state the results here for simple endofunctors and (co)algebras, but they generalize to parameterized endofunctors and (co)algebras.

Theorem D (Characterization of initial algebras and terminal coalgebras in Σ -categories, Corollary 49 and Theorem 52). *Let E be a split fibration endofunctor on $\Sigma_C\mathcal{L}$ (resp. $\Sigma_C\mathcal{L}^{op}$) and let (\overline{E}, e) be the corresponding strictly indexed endofunctor on \mathcal{L} . Then, E has a (fibered) initial algebra if*

- $\overline{E} : \mathcal{C} \rightarrow \mathcal{C}$ has an initial algebra $\text{in}_{\overline{E}} : \overline{E}(\mu\overline{E}) \rightarrow \mu\overline{E}$;
- $\mathcal{L}(\text{in}_{\overline{E}})^{-1} e_{\mu\overline{E}} : \mathcal{L}(\mu\overline{E}) \rightarrow \mathcal{L}(\mu\overline{E})$ has an initial algebra (resp. terminal coalgebra);
- $\mathcal{L}(f)$ preserves initial algebras (resp. terminal coalgebras) for all morphisms $f \in \mathcal{C}$;

and E has a (fibered) terminal coalgebra if

- $\overline{E} : \mathcal{C} \rightarrow \mathcal{C}$ has a terminal coalgebra $\text{out}_{\overline{E}} : \nu\overline{E} \rightarrow \overline{E}(\nu\overline{E})$;
- $\mathcal{L}(\text{out}_{\overline{E}}) e_{\nu\overline{E}} : \mathcal{L}(\nu\overline{E}) \rightarrow \mathcal{L}(\nu\overline{E})$ has a terminal coalgebra (resp. initial algebra)
- $\mathcal{L}(f)$ preserves terminal coalgebras (resp. initial algebras) for all morphisms $f \in \mathcal{C}$.

We use this result to give sufficient conditions for (fibered) $\mu\nu$ -polynomials (including their fibered initial algebras and terminal coalgebras) to exist in $\Sigma_C\mathcal{L}$ and $\Sigma_C\mathcal{L}^{op}$. In particular, we show that it suffices to extend the target language $\mathbf{LSyn} : \mathbf{CSyn}^{op} \rightarrow \mathbf{Cat}$ with both cartesian and linear inductive and coinductive types to perform CHAD on a source language \mathbf{Syn} with inductive and coinductive types. Again, an equivalent of Theorem B holds.

We write $\mathbf{roll} x$ for the constructor of inductive types (applied to an identifier x), $\mathbf{unroll} x$ for the destructor of coinductive types, and $\tau.\mathbf{roll}^{-1} x \stackrel{\text{def}}{=} \mathbf{fold} x \mathbf{with} y \rightarrow \tau[y^{\mathbf{roll}} y/\alpha]$, where we write $\tau[y^{\mathbf{roll}} y/\alpha]$ for the functorial action of the parameterized type τ with type parameter α on the term $\mathbf{roll} y$ in context y . This yields the following formula for spaces of primals and (co)tangent vectors to (co)inductive types where:

$$\begin{array}{ll} \vec{\mathcal{D}}(\alpha)_1 \stackrel{\text{def}}{=} \alpha & \vec{\mathcal{D}}(\alpha)_2 = \underline{\alpha} \\ \vec{\mathcal{D}}\mu\alpha.(\tau)_1 \stackrel{\text{def}}{=} \mu\underline{\alpha}.\vec{\mathcal{D}}(\tau)_1 & \vec{\mathcal{D}}\mu\alpha.(\tau)_2 \stackrel{\text{def}}{=} \underline{\mu\underline{\alpha}}.\vec{\mathcal{D}}(\tau)_2[\vec{\mathcal{D}}(\tau)_1.\mathbf{roll}^{-1}p/p] \\ \vec{\mathcal{D}}\nu\alpha.(\tau)_1 \stackrel{\text{def}}{=} \nu\underline{\alpha}.\vec{\mathcal{D}}(\tau)_1 & \vec{\mathcal{D}}\nu\alpha.(\tau)_2 \stackrel{\text{def}}{=} \underline{\nu\underline{\alpha}}.\vec{\mathcal{D}}(\tau)_2[\mathbf{unroll} p/p] \\ \overleftarrow{\mathcal{D}}(\alpha)_1 \stackrel{\text{def}}{=} \alpha & \overleftarrow{\mathcal{D}}(\alpha)_2 = \underline{\alpha} \\ \overleftarrow{\mathcal{D}}(\mu\alpha.(\tau))_1 \stackrel{\text{def}}{=} \mu\underline{\alpha}.\overleftarrow{\mathcal{D}}(\tau)_1 & \overleftarrow{\mathcal{D}}(\mu\alpha.(\tau))_2 \stackrel{\text{def}}{=} \underline{\nu\underline{\alpha}}.\overleftarrow{\mathcal{D}}(\tau)_2[\vec{\mathcal{D}}(\tau)_1.\mathbf{roll}^{-1}p/p] \\ \overleftarrow{\mathcal{D}}(\nu\alpha.(\tau))_1 \stackrel{\text{def}}{=} \nu\underline{\alpha}.\overleftarrow{\mathcal{D}}(\tau)_1 & \overleftarrow{\mathcal{D}}(\nu\alpha.(\tau))_2 \stackrel{\text{def}}{=} \underline{\mu\underline{\alpha}}.\overleftarrow{\mathcal{D}}(\tau)_2[\mathbf{unroll} p/p] \end{array}$$

Insight 6. Types of primals to (co)inductive types are (co)inductive types of primals, types of tangents to (co)inductive types are linear (co)inductive types of tangents, and types of cotangents to inductive types are linear coinductive types of cotangents and vice versa.

For example, for a type $\tau = \mu\alpha. \{ \mathbf{Empty} \mathbf{1} \mid \mathbf{Cons}(\sigma * \alpha) \}$ of lists of elements of type σ , we have a cotangent space:

$$\overleftarrow{\mathcal{D}}(\tau)_2 = \underline{\nu\underline{\alpha}}.\mathbf{case} \mathbf{roll}^{-1} p \mathbf{of} \{ \mathbf{Empty} _ \rightarrow \mathbf{1} \mid \mathbf{Cons} p \rightarrow \overleftarrow{\mathcal{D}}(\sigma)_2[\mathbf{fst} p/p] * \underline{\alpha} \} \quad \text{where}$$

$$\mathbf{roll}^{-1} p = \mathbf{fold} p \mathbf{with} y \rightarrow \mathbf{case} y \mathbf{of} \{ \mathbf{Empty} y \rightarrow \mathbf{Empty} y \mid \mathbf{Cons} y \rightarrow \mathbf{Cons}(\mathbf{fst} y, \mathbf{roll}(\mathbf{snd} y)) \}$$

and, for a type $\tau = \nu\alpha.\sigma * \alpha$ of streams, we have a cotangent space:

$$\overleftarrow{\mathcal{D}}(\tau)_2 = \underline{\mu\underline{\alpha}}.\overleftarrow{\mathcal{D}}(\sigma)_2[\mathbf{fst}(\mathbf{unroll} p)/p] * \underline{\alpha}.$$

We demonstrate that the strictly indexed category $\mathbf{FVect} : \mathbf{Set}^{op} \rightarrow \mathbf{Cat}$ of families of vector spaces also satisfies our conditions, so it gives a concrete denotational semantics of the target language $\mathbf{LSyn} : \mathbf{CSyn}^{op} \rightarrow \mathbf{Cat}$, by Theorem B. To reestablish the correctness Theorem A, existing logical relations techniques do not suffice, as far as we are aware. Instead, we achieve it by developing a novel theory of categorical logical relations (sconing) for languages with expressive type systems like our AD source language.

Insight 7. We can obtain powerful logical relations techniques for reasoning about expressive type systems by analyzing when the forgetful functor from a category of logical relations to the underlying category is comonadic and monadic.

In almost all instances, the forgetful functor from a category of logical relations to the underlying category is comonadic and in many instances, including ours, it is even monadic. This gives us the following logical relations techniques for expressive type systems:

Theorem E (Logical relations for expressive types, Section 11). *Let $G : \mathcal{C} \rightarrow \mathcal{D}$ be a functor. We observe*

- If \mathcal{D} has binary products, then the forgetful functor from the scone (the comma category) $\mathcal{D} \downarrow G \rightarrow \mathcal{D} \times \mathcal{C}$ is comonadic (Theorem 97).
- If G has a left adjoint and \mathcal{C} has binary coproducts, then $\mathcal{D} \downarrow G \rightarrow \mathcal{D} \times \mathcal{C}$ is monadic (Corollary 99).

This is relevant because:

- comonadic functors create initial algebras (Theorem 109);
- monadic functors create terminal coalgebras (Theorem 109);
- monadic-comonadic functors create $\mu\nu$ -polynomials (Corollary 110);
- if \mathcal{E} is monadic-comonadic over \mathcal{E}' , then \mathcal{E} is finitely complete cartesian closed if \mathcal{E}' is (Proposition 103).

As a consequence, we can lift our concrete denotational semantics of all types, including inductive and coinductive types to our categories of logical relations over the semantics.

These logical relations techniques are sufficient to yield the correctness Theorem A. Indeed, as long as derivatives of primitive operations are correctly implemented in the sense that $\llbracket \text{Dop} \rrbracket = \text{Dop}$ and $\llbracket \text{Dop}^t \rrbracket = D[\text{op}]^t$, Theorem E tells us that the unique structure-preserving functors:

$$\begin{aligned} (\llbracket - \rrbracket, \llbracket \overrightarrow{\mathcal{D}}(-) \rrbracket) : \mathbf{Syn} &\rightarrow \mathbf{Set} \times \Sigma_{\mathbf{Set}} \mathbf{FVect} \\ (\llbracket - \rrbracket, \llbracket \overleftarrow{\mathcal{D}}(-) \rrbracket) : \mathbf{Syn} &\rightarrow \mathbf{Set} \times \Sigma_{\mathbf{Set}} \mathbf{FVect}^{op} \end{aligned}$$

lift to the scones of $\text{Hom}((\mathbb{R}^k, (\mathbb{R}^k, \underline{\mathbb{R}}^k)), -) : \mathbf{Set} \times \Sigma_{\mathbf{Set}} \mathbf{FVect} \rightarrow \mathbf{Set}$ and $\text{Hom}((\mathbb{R}^k, (\mathbb{R}^k, \underline{\mathbb{R}}^k)), -) : \mathbf{Set} \times \Sigma_{\mathbf{Set}} \mathbf{FVect}^{op} \rightarrow \mathbf{Set}$ where we lift the image of \mathbf{real}^n , respectively, to the logical relations:

$$\begin{aligned} \{ (f, (g, h)) \mid f = g \text{ and } h = Df \} &\hookrightarrow (\mathbf{Set} \times \Sigma_{\mathbf{Set}} \mathbf{FVect}) \left((\mathbb{R}^k, (\mathbb{R}^k, \underline{\mathbb{R}}^k)), (\mathbb{R}^n, (\mathbb{R}^n, \underline{\mathbb{R}}^n)) \right) \\ \{ (f, (g, h)) \mid f = g \text{ and } h = Df^t \} &\hookrightarrow (\mathbf{Set} \times \Sigma_{\mathbf{Set}} \mathbf{FVect}^{op}) \left((\mathbb{R}^k, (\mathbb{R}^k, \underline{\mathbb{R}}^k)), (\mathbb{R}^n, (\mathbb{R}^n, \underline{\mathbb{R}}^n)) \right). \end{aligned}$$

We see that $\llbracket \overrightarrow{\mathcal{D}}(t) \rrbracket$ and $\llbracket \overleftarrow{\mathcal{D}}(t) \rrbracket$ propagate derivatives and transposed derivatives of differentiable k -surfaces (differentiable functions $\mathbb{R}^k \rightarrow \text{dom}[\llbracket t \rrbracket]$) correctly for all programs t . Seeing that $(\text{id}, (\text{id}, x \mapsto \text{id}))$ is one such k -surface in the logical relation associated with \mathbf{real}^k , we see that $(\llbracket t \rrbracket, (\pi_1 \circ \llbracket \overrightarrow{\mathcal{D}}(t) \rrbracket, \pi_2 \circ \llbracket \overrightarrow{\mathcal{D}}(t) \rrbracket))$ and $(\llbracket t \rrbracket, (\pi_1 \circ \llbracket \overleftarrow{\mathcal{D}}(t) \rrbracket, \pi_2 \circ \llbracket \overleftarrow{\mathcal{D}}(t) \rrbracket))$ are k -surfaces in the relations as well, for any $x : \mathbf{real}^k \vdash t : \mathbf{real}^n$. That is, Theorem A holds.

Our novel logical relations machinery is in no way restricted to the context of CHAD, however. In fact, it is widely applicable for reasoning about total functional languages with expressive type systems.

2.6 Inductive types and derivatives

So far, we have only phrased the CHAD correctness Theorem A only for programs t with domain/codomain isomorphic to some Euclidean space \mathbb{R}^n , even if t may make use of any complex types (including variant, inductive, coinductive, and function types) in its computation. The reason for this restriction is that this limited context of functions $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$ is an obvious setting where we have a simple, canonical, unambiguous notion of derivative $\mathcal{T}f : \mathbb{R}^n \rightarrow \mathbb{R}^m \times (\mathbb{R}^n \multimap \mathbb{R}^m)$, allowing us to phrase an obvious correctness criterion.

More generally, for $f : X \rightarrow Y$ where X and Y are manifolds, we also have an unambiguous notion of derivative $\mathcal{T}f : \prod_{x \in X} \Sigma_{y \in Y} \mathcal{T}_x X \multimap \mathcal{T}_y Y$, which allows us to strengthen our correctness result. In fact, for our purposes, it suffices to consider the relatively simple context of differentiable functions $f : \coprod_{i \in I} \mathbb{R}^{n_i} \rightarrow \coprod_{j \in J} \mathbb{R}^{m_j}$ between very simple manifolds that arise as disjoint unions of

(finite-dimensional) Euclidean spaces. Such functions f decompose uniquely as copairings $f = [\iota_{\phi(i)} \circ g_i]_{i \in I}$ where we write ι_k for the k -th coprojection and where $\phi : I \rightarrow J$ is some function and $g_i : \mathbb{R}^{n_i} \rightarrow \mathbb{R}^{m_{\phi(i)}}$. That is, f can be understood as the family $(g_i)_{i \in I}$ and its derivative $\mathcal{T}f$ decomposes uniquely as the family of plain derivatives \mathcal{T}_{g_i} in the usual sense. We have a similar decomposition for the transposed derivatives \mathcal{T}^*f .

This notion of derivatives of functions between disjoint unions of Euclidean spaces is relevant to our context, as we have the following result.

Theorem F (Canonical form of μ -polynomial semantics, Corollary 127). *For any types τ_i built from Euclidean spaces \mathbf{real}^n , tuple types $\tau_i * \tau_j$, variant types $\{\ell_1 \tau_1 \mid \dots \mid \ell_n \tau_n\}$, type variables α , and inductive types $\mu \alpha. \tau_i$ (so-called μ -polynomials), its denotation $\llbracket \tau_i \rrbracket$ is isomorphic to a manifold of the form $\coprod_{i \in I} \mathbb{R}^{n_i}$ for some countable set I and some $n_i \in \mathbb{N}$.*

Consequently, we can strengthen Theorem A in the following form:

Theorem G (Correctness of CHAD (Generalized), Theorem 129). *For any well-typed program*

$$x_1 : \tau_1, \dots, x_k : \tau_n \vdash t : \sigma,$$

where τ_i, σ are all (closed) μ -polynomials, we have that $\llbracket \overrightarrow{\mathcal{D}}_{x_1, \dots, x_k}(t) \rrbracket = \mathcal{T}_{\llbracket t \rrbracket}$ and $\llbracket \overleftarrow{\mathcal{D}}_{x_1, \dots, x_k}(t) \rrbracket = \mathcal{T}^* \llbracket t \rrbracket$.

Again, t can make use of coinductive types and function types in the middle of its computation, but they may not occur in the input or output types. The reason is that, as far as we are aware, there is no canonical³ notion of semantic derivative for functions between the sort of infinite-dimensional spaces that co-datatypes such as coinductive types and function types implement. This makes it challenging to even phrase what semantic correctness at such types would mean.

2.7 How does CHAD for expressive types work in practice?

The CHAD code transformations we describe in this paper are well behaved in practical implementations in the sense of the following compile-time complexity result.

Theorem H (No code blowup, Corollary 130). *The size of the code of the CHAD transformed programs $\overrightarrow{\mathcal{D}}_{\overline{\tau}}(t)$ and $\overleftarrow{\mathcal{D}}_{\overline{\tau}}(t)$ grows linearly with the size of the original source program t .*

We have ensured to pair up the primal and (co)tangent computations in our CHAD transformation and to exploit any possible sharing of common subcomputations, using **let**-bindings. However, we leave a formal study of the runtime complexity of our technique to future work.

As formulated in this paper, CHAD generates code with linear dependent types. This seems very hard to implement in practice. However, this is an illusion: we can use the code generated by CHAD and interpret it as less precise types. We sketch how all type dependency can be erased and how all linear types other than the linear (co)inductive types can be implemented as abstract types in a standard functional language like Haskell. In fact, we describe three practical implementation strategies for our treatment of sum types, none of which require linear or dependent types. All three strategies have been shown to work in the CHAD reference implementation. We suggest how linear (co)inductive types might be implemented in practice, based on their concrete denotational semantics, but leave the actual implementation to future work.

3. Background: Categorical Semantics of Expressive Total Languages

In this section, we fix some notation and recall the well-known abstract categorical semantics of total functional languages with expressive type systems (Crole 1993; Pitts 1995; Santocanale 2002), which builds on the usual semantics of the simply typed λ -calculus in Cartesian closed categories

(Lambek and Scott 1988). In this paper, we will be interested in a few particular instantiations (or models) of such an abstract categorical semantics \mathcal{C} :

- the initial model **Syn** (Section 5), which represents the programming language under consideration, up to $\beta\eta$ -equivalence; this will be the source language of our AD code transformation;
- the concrete denotational model **Set** (Section 9) in terms of sets and functions, which represents our default denotational semantics of the source language;
- models $\Sigma_{\mathcal{C}}\mathcal{L}$ and $\Sigma_{\mathcal{C}}\mathcal{L}^{op}$ (Section 6) in the the Σ -types of suitable indexed categories $\mathcal{L} : \mathcal{C}^{op} \rightarrow \mathbf{Cat}$;
- in particular, the models $\Sigma_{\mathbf{CSyn}}\mathbf{LSyn}$ and $\Sigma_{\mathbf{CSyn}}\mathbf{LSyn}^{op}$ (Section 7) built out of the target language, which yield forward and reverse-mode CHAD code transformations, respectively;
- scoping (categorical logical relations) constructions $\overleftarrow{\mathbf{Scone}}$ and $\overrightarrow{\mathbf{Scone}}$ (Section 11) over the models $\mathbf{Set} \times \Sigma_{\mathbf{Set}}\mathbf{FVect}$ and $\mathbf{Set} \times \Sigma_{\mathbf{Set}}\mathbf{FVect}^{op}$ that yield the correctness arguments for forward- and reverse-mode CHAD, respectively, where $\mathbf{FVect} : \mathbf{Set}^{op} \rightarrow \mathbf{Cat}$ is the strictly indexed category of families of real vector spaces.

We deem it relevant to discuss the abstract categorical semantic framework for our language as we need these various instantiations of the framework.

3.1 Basics

We use standard definitions from category theory; see, for instance, Mac Lane (1971), Leinster (2014). A category \mathcal{C} can be seen as a semantics for a typed functional programming language, whose types correspond to objects of \mathcal{C} and whose programs that take an input of type A and produce an output of type B are represented by the homset $\mathcal{C}(A, B)$. Identity morphisms id_A represent programs that simply return their input (of type A) unchanged as output and composition $g \circ f$ of morphisms f and g represents running the program g after the program f . Notably, the equations that hold between morphisms represent program equivalences that hold for the particular notion of semantics that \mathcal{C} represents. Some of these program equivalences are so fundamental that we demand them as structural equalities that need to hold in any categorical model (such as the associativity law $f \circ (g \circ h) = (f \circ g) \circ h$). In programming languages terms, these are known as the β - and η -equivalences of programs.

3.2 Tuple types

Tuple types represent a mechanism for letting programs take more than one input or produce more than one output. Categorically, a tuple type corresponds to a product $\prod_{i \in I} A_i$ of a finite family of types $\{A_i\}_{i \in I}$, which we also write $\mathbb{1}$ or $A_1 \times A_2$ in the case of nullary and binary products. For basic aspects of products, we refer the reader to Mac Lane (1971, Chapter III).

We write $(f_i)_{i \in I} : C \rightarrow \prod_{i \in I} A_i$ for the product pairing of $\{f_i : C \rightarrow A_i\}_{i \in I}$ and $\pi_j : \prod_{i \in I} A_i \rightarrow A_j$ for the j -th product projection, for $j \in I$. As such, we say that a categorical semantics \mathcal{C} models (finite) tuples if \mathcal{C} has (chosen) finite products.

3.3 Primitive types and operations

We are interested in programming languages that have support for a certain set Ty of ground types such as integers and (floating point) real numbers as well as certain sets $\text{Op}(T_1, \dots, T_n; S)$, for $T_1, \dots, T_n, S \in \text{Ty}$, of operations on these basic types such as addition, multiplication, and sine functions. We model such primitive types and operations categorically by demanding that our category has a distinguished object C_T for each $T \in \text{Ty}$ to represent the primitive types

and a distinguished morphism $f_{op} \in \mathcal{C}(C_{T_1} \times \dots \times C_{T_n}, C_S)$ for all primitive operations $op \in \text{Op}(T_1, \dots, T_n; S)$. For basic aspects of categorical type theory, see, for instance, Crole (1993, Chapters 3&4).

3.4 Function types

Function types let us type popular higher-order programming idioms such as maps and folds, which capture common control flow abstractions. Categorically, a type of functions from A to B is modeled as an exponential $A \Rightarrow B$. We write $ev : (A \Rightarrow B) \times A \rightarrow B$ (evaluation) for the counit of the adjunction $(-) \times A \dashv A \Rightarrow (-)$ and Λ for the Curryng natural isomorphism $\mathcal{C}(A \times B, C) \rightarrow \mathcal{C}(A, B \Rightarrow C)$. We say that a categorical semantics \mathcal{C} with tuple types models function types if \mathcal{C} has a chosen right adjoint $(-) \times A \dashv A \Rightarrow (-)$.

3.5 Sum types (aka variant types)

Sum types (aka variant types) let us model data that exists in multiple different variants and branch in our code on these different possibilities. Categorically, a sum type is modeled as a coproduct $\coprod_{i \in I} A_i$ of a collection of a finite family $\{A_i\}_{i \in I}$ of types, which we also write 0 or $A_1 \sqcup A_2$ in the case of nullary and binary coproducts. We write $[f_i]_{i \in I} : \coprod_{i \in I} C_i \rightarrow A$ for the copairing of $\{f_i : C_i \rightarrow A\}_{i \in I}$ and $\iota_j : A_j \rightarrow \coprod_{i \in I} A_i$ for the j -th coprojection. In fact, in presence of tuple types, a more useful programming interface is obtained if one restricts to *distributive* coproducts, that is, coproducts $\coprod_{i \in I} A_i$ such that the map $[(\iota_i \circ \pi_1) \pi_2]_{i \in I} : \coprod_{i \in I} (A_i \times B) \rightarrow (\coprod_{i \in I} A_i) \times B$ is an isomorphism; see, for instance, Carboni et al. (1993), Lack (2012). Note that in presence of function types, coproducts are automatically distributive since the left adjoint functors $(-) \times A$ preserve colimits; see, for instance, Leinster (2014, 6.3). As such, we say that a categorical semantics \mathcal{C} models (finite) sum types if \mathcal{C} has (chosen) finite distributive coproducts.

3.6 Inductive and coinductive types

We employ the usual semantic interpretation of *inductive and coinductive types* as, respectively, *initial algebras* and *terminal coalgebras* of a certain class of functors. We refer the reader, for instance, to Barr and Wells (2005, Chapter 9), Santocanale (2002), and Adamek et al. (2010).

Most of this section is dedicated to describing precisely which class of functors we consider initial algebras and terminal coalgebras, a class we call $\mu\nu$ -polynomials. Roughly speaking, we define $\mu\nu$ -polynomials to be functors that can be constructed from products, coproducts, projections, diagonals, constants, initial algebras, and terminal coalgebras.

To fix terminology and for future reference of the detailed constructions, we recall below basic aspects of parameterized initial algebras and parameterized terminal coalgebras.

Definition 1 (The category of E -algebras). *Let $E : \mathcal{D} \rightarrow \mathcal{D}$ be an endofunctor. The category of E -algebras, denoted by $E\text{-Alg}$, is defined as follows. The objects are pairs (W, ζ) in which $W \in \mathcal{D}$ and $\zeta : E(W) \rightarrow W$ is a morphism of \mathcal{D} . A morphism between E -algebras (W, ζ) and (Y, ξ) is a morphism $g : W \rightarrow Y$ of \mathcal{D} such that*

$$\begin{array}{ccc}
 E(W) & \xrightarrow{E(g)} & E(Y) \\
 \zeta \downarrow & & \downarrow \xi \\
 W & \xrightarrow{g} & Y
 \end{array} \tag{1}$$

commutes. Dually, we define the category $E\text{-CoAlg}$ of E -coalgebras by:

$$E\text{-CoAlg} := (E^{\text{op}}\text{-Alg})^{\text{op}} \tag{2}$$

in which $E^{\text{op}} : \mathcal{D}^{\text{op}} \rightarrow \mathcal{D}^{\text{op}}$ is the image of E by $\text{op} : \mathbf{Cat} \rightarrow \mathbf{Cat}$.

Definition 2 (Initial algebra and terminal coalgebra). Let $E : \mathcal{D} \rightarrow \mathcal{D}$ be an endofunctor. Provided that they exist, the initial object $(\mu E, \text{in}_E)$ of $E\text{-Alg}$ and the terminal object $(\nu E, \text{out}_E)$ of $E\text{-CoAlg}$ are, respectively, referred to as the initial E -algebra and the terminal E -coalgebra.

Remark 3. By Lambek’s Theorem, provided that the initial algebra $(\mu E, \text{in}_E)$ of an endofunctor E exists, we have that in_E is invertible. Dually, we get the result for terminal coalgebras.

Assuming the existence of the initial E -algebra and the terminal E -coalgebra, we denote by:

$$\text{fold}_E(Y, \xi) : \mu E \rightarrow Y, \quad \text{unfold}_E(X, \varrho) : X \rightarrow \nu E \tag{3}$$

the unique morphisms in \mathcal{D} such that

$$\begin{array}{ccc} E(\mu E) & \xrightarrow{E(\text{fold}_E(Y, \xi))} & E(Y) \\ \text{in}_E \downarrow & & \downarrow \xi \\ \mu E & \xrightarrow{\text{fold}_E(Y, \xi)} & Y \end{array} \quad \begin{array}{ccc} X & \xrightarrow{\text{unfold}_E(X, \varrho)} & \nu E \\ \varrho \downarrow & & \downarrow \text{out}_E \\ E(X) & \xrightarrow{E(\text{unfold}_E(X, \varrho))} & E(\nu E) \end{array} \tag{4}$$

commute. Whenever it is clear from the context, we denote $\text{fold}_E(Y, \xi)$ by $\text{fold}_E \xi$, and $\text{unfold}_E(X, \varrho)$ by $\text{unfold}_E \varrho$.

Given a functor $H : \mathcal{D}' \times \mathcal{D} \rightarrow \mathcal{D}$ and an object X of \mathcal{D}' , we denote by H^X the endofunctor:

$$H(X, -) : \mathcal{D} \rightarrow \mathcal{D}. \tag{5}$$

In this setting, if μH^X exists for any object $X \in \mathcal{D}'$ then the universal properties of the initial algebras induce a functor denoted by $\mu H : \mathcal{D}' \rightarrow \mathcal{D}$, called the parameterized initial algebra. In the following, we spell out how to construct parameterized initial algebras and terminal coalgebras.

Proposition 4 (μ -operator and ν -operator). Let $H : \mathcal{D}' \times \mathcal{D} \rightarrow \mathcal{D}$ be a functor. Assume that, for each object $X \in \mathcal{D}'$, the functor $H^X = H(X, -)$ is such that μH^X exists. In this setting, we have the induced functor:

$$\begin{aligned} \mu H : \mathcal{D}' &\rightarrow \mathcal{D} \\ X &\mapsto \mu H^X \\ (f : X \rightarrow Y) &\mapsto \text{fold}_{H^X} (\text{in}_{H^Y} \circ H(f, \mu H^Y)). \end{aligned}$$

Dually, assuming that, for each object $X \in \mathcal{D}'$, νH^X exists, we have the induced functor:

$$\begin{aligned} \nu H : \mathcal{D}' &\rightarrow \mathcal{D} \\ X &\mapsto \nu H^X \\ (f : X \rightarrow Y) &\mapsto \text{unfold}_{H^Y} (H(f, \nu H^X) \circ \text{out}_{H^X}). \end{aligned}$$

Proof. We assume that the functor $H : \mathcal{D}' \times \mathcal{D} \rightarrow \mathcal{D}$ is such that, for any object $X \in \mathcal{D}'$, μH^X exists. For each morphism $f : X \rightarrow Y$, we define $\mu H(f) = \text{fold}_{H^X} (\text{in}_{H^Y} \circ H(f, \mu H^Y))$ as above. We prove below that this makes $\mu H(f)$ a functor.

Given $X \in \mathcal{D}'$,

$$\begin{aligned} &\mu H(\text{id}_X) \\ &= \text{fold}_{H^X} (\text{in}_{H^X} \circ H(\text{id}_X, \mu H^X)) \\ &= \text{fold}_{H^X} (\text{in}_{H^X}) \\ &= \text{id}_{\mu H^X}. \end{aligned}$$

Moreover, given morphisms $f : X \rightarrow Y$ and $g : Y \rightarrow Z$ in \mathcal{D}' , we have that

$$\begin{aligned} &\mu H(g) \circ \mu H(f) \circ \text{in}_{H^X} \\ &= \mu H(g) \circ \text{in}_{H^Y} \circ H(f, \mu H(f)) && \{ \mu H(f) = \text{fold}_{H^Y} (\text{in}_{H^Y} \circ H(f, \mu H^Y)) \} \\ &= \text{in}_{H^Z} \circ H(g, \mu H(g)) \circ H(f, \mu H(f)) && \{ \mu H(g) = \text{fold}_{H^Z} (\text{in}_{H^Z} \circ H(g, \mu H^Z)) \} \\ &= \text{in}_{H^Z} \circ H(gf, \mu H(g) \circ \mu H(f)) \\ &= \text{in}_{H^Z} \circ H(gf, \mu H^Z) \circ H(X, \mu H(g) \circ \mu H(f)) \end{aligned}$$

and, hence, the diagram:

$$\begin{array}{ccc} H(X, \mu H^X) & \xrightarrow{H(X, \mu H(g) \circ \mu H(f))} & H(X, \mu H^Z) \\ \text{in}_{H^X} \downarrow & & \downarrow \text{in}_{H^Z} \circ H(g \circ f, \mu H^Z) \\ \mu H^X & \xrightarrow{\mu H(g) \circ \mu H(f)} & \mu H^Z \end{array}$$

commutes. By the universal property of the initial algebra $(\mu H^X, \text{in}_{H^X})$, we conclude that

$$\begin{aligned} &\mu H(g) \circ \mu H(f) \\ &= \text{fold}_{H^X} (\text{in}_{H^Z} \circ H(g \circ f, \mu H^Z)) \\ &= \mu H(g \circ f). \end{aligned} \qquad \{ \text{definition} \} \quad \square$$

It is worth noting that in Proposition 4, \mathcal{D}' can be any category. However, in the standard setting of initial algebra semantics, there is a special interest in the case where $\mathcal{D}' = \mathcal{D}^{n-1}$ and $n > 1$, which is described below.

Proposition 5 (Parameterized initial algebras and terminal coalgebras). *Let $H : \mathcal{D}^n \rightarrow \mathcal{D}$ be a functor in which $n > 1$. Assume that, for each object $X \in \mathcal{D}^{n-1}$, μH^X exists. In this setting, we have the induced functor:*

$$\begin{aligned} &\mu H : \mathcal{D}^{n-1} \rightarrow \mathcal{D} \\ &X \mapsto \mu H^X \\ &(f : X \rightarrow Y) \mapsto \text{fold}_{H^X} (\text{in}_{H^Y} \circ H(f, \mu H^Y)). \end{aligned}$$

Dually, if νH^X exists for any $X \in \mathcal{D}^{n-1}$, we have the induced functor:

$$\begin{aligned} &\nu H : \mathcal{D}^{n-1} \rightarrow \mathcal{D} \\ &X \mapsto \nu H^X \\ &(f : X \rightarrow Y) \mapsto \text{unfold}_{H^Y} (H(f, \nu H^X) \circ \text{out}_{H^X}). \end{aligned}$$

In order to model inductive and coinductive types coming from parameterized types not involving function types, we introduce the following notions.

Definition 6 ($\mu\nu$ -polynomials). Assuming that \mathcal{D} has finite coproducts and finite products, the category $\mu\nu\text{Poly}_{\mathcal{D}}$ is the smallest subcategory of \mathbf{Cat} satisfying the following.

(O) The objects are defined inductively by:

(O1) the terminal category $\mathbb{1}$ is an object of $\mu\nu\text{Poly}_{\mathcal{D}}$;

(O2) the category \mathcal{D} is an object of $\mu\nu\text{Poly}_{\mathcal{D}}$;

(O3) for any pair of objects $(\mathcal{D}', \mathcal{D}'') \in \mu\nu\text{Poly}_{\mathcal{D}} \times \mu\nu\text{Poly}_{\mathcal{D}}$, the product $\mathcal{D}' \times \mathcal{D}''$ is an object of $\mu\nu\text{Poly}_{\mathcal{D}}$.

(M) The morphisms satisfy the following properties:

(M1) for any object \mathcal{D}' of $\mu\nu\text{Poly}_{\mathcal{D}}$, the unique functor $\mathcal{D}' \rightarrow \mathbb{1}$ is a morphism of $\mu\nu\text{Poly}_{\mathcal{D}}$;

(M2) for any object \mathcal{D}' of $\mu\nu\text{Poly}_{\mathcal{D}}$, all the functors $\mathbb{1} \rightarrow \mathcal{D}'$ are morphisms of $\mu\nu\text{Poly}_{\mathcal{D}}$;

(M3) the binary product $\times : \mathcal{D} \times \mathcal{D} \rightarrow \mathcal{D}$ is a morphism of $\mu\nu\text{Poly}_{\mathcal{D}}$;

(M4) the binary coproduct $\sqcup : \mathcal{D} \times \mathcal{D} \rightarrow \mathcal{D}$ is a morphism of $\mu\nu\text{Poly}_{\mathcal{D}}$;

(M5) for any pair of objects $(\mathcal{D}', \mathcal{D}'') \in \mu\nu\text{Poly}_{\mathcal{D}} \times \mu\nu\text{Poly}_{\mathcal{D}}$, the projections:

$$\pi_1 : \mathcal{D}' \times \mathcal{D}'' \rightarrow \mathcal{D}', \quad \pi_2 : \mathcal{D}' \times \mathcal{D}'' \rightarrow \mathcal{D}''$$

are morphisms of $\mu\nu\text{Poly}_{\mathcal{D}}$;

(M6) given objects $\mathcal{D}', \mathcal{D}'', \mathcal{D}'''$ of $\mu\nu\text{Poly}_{\mathcal{D}}$, if $E : \mathcal{D}' \rightarrow \mathcal{D}''$ and $J : \mathcal{D}' \rightarrow \mathcal{D}'''$ are morphisms of $\mu\nu\text{Poly}_{\mathcal{D}}$, then so is the induced functor $(E, J) : \mathcal{D}' \rightarrow \mathcal{D}'' \times \mathcal{D}'''$;

(M7) if \mathcal{D}' is an object of $\mu\nu\text{Poly}_{\mathcal{D}}$, $H : \mathcal{D}' \times \mathcal{D} \rightarrow \mathcal{D}$ is a morphism of $\mu\nu\text{Poly}_{\mathcal{D}}$ and $\mu H : \mathcal{D}' \rightarrow \mathcal{D}$ exists, then μH is a morphism of $\mu\nu\text{Poly}_{\mathcal{D}}$;

(M8) if \mathcal{D}' is an object of $\mu\nu\text{Poly}_{\mathcal{D}}$, $H : \mathcal{D}' \times \mathcal{D} \rightarrow \mathcal{D}$ is a morphism of $\mu\nu\text{Poly}_{\mathcal{D}}$ and $\nu H : \mathcal{D}' \rightarrow \mathcal{D}$ exists, then νH is a morphism of $\mu\nu\text{Poly}_{\mathcal{D}}$.

We say that \mathcal{D} has $\mu\nu$ -polynomials if \mathcal{D} has finite coproducts and products and, for any endofunctor $E : \mathcal{D} \rightarrow \mathcal{D}$ in $\mu\nu\text{Poly}_{\mathcal{D}}$, μE and νE exist. We say that \mathcal{D} has chosen $\mu\nu$ -polynomials if we have additionally made a choice of initial algebras and terminal coalgebras for all $\mu\nu$ -polynomials.

Remark 7 (Self-duality). A category \mathcal{D} has $\mu\nu$ -polynomials if and only if \mathcal{D}^{op} has $\mu\nu$ -polynomials as well.

Another suitably equivalent way of defining $\mu\nu\text{Poly}_{\mathcal{D}}$ is the following. The category $\mu\nu\text{Poly}_{\mathcal{D}}$ is the smallest subcategory of \mathbf{Cat} such that:

- the inclusion $\mu\nu\text{Poly}_{\mathcal{D}} \rightarrow \mathbf{Cat}$ creates finite products;
- \mathcal{D} is an object of the subcategory $\mu\nu\text{Poly}_{\mathcal{D}}$;
- for any object \mathcal{D}' of $\mu\nu\text{Poly}_{\mathcal{D}}$, all the functors $\mathbb{1} \rightarrow \mathcal{D}'$ are morphisms of $\mu\nu\text{Poly}_{\mathcal{D}}$;
- and the binary product $\times : \mathcal{D} \times \mathcal{D} \rightarrow \mathcal{D}$ is a morphism of $\mu\nu\text{Poly}_{\mathcal{D}}$;
- the binary coproduct $\sqcup : \mathcal{D} \times \mathcal{D} \rightarrow \mathcal{D}$ is a morphism of $\mu\nu\text{Poly}_{\mathcal{D}}$;
- if \mathcal{D}' is an object of $\mu\nu\text{Poly}_{\mathcal{D}}$, $H : \mathcal{D}' \times \mathcal{D} \rightarrow \mathcal{D}$ is a morphism of $\mu\nu\text{Poly}_{\mathcal{D}}$ and $\mu H : \mathcal{D}' \rightarrow \mathcal{D}$ exists, then μH is a morphism of $\mu\nu\text{Poly}_{\mathcal{D}}$;
- if \mathcal{D}' is an object of $\mu\nu\text{Poly}_{\mathcal{D}}$, $H : \mathcal{D}' \times \mathcal{D} \rightarrow \mathcal{D}$ is a morphism of $\mu\nu\text{Poly}_{\mathcal{D}}$ and $\nu H : \mathcal{D}' \rightarrow \mathcal{D}$ exists, then νH is a morphism of $\mu\nu\text{Poly}_{\mathcal{D}}$.

Lemma 8. Let \mathcal{C} be a category with $\mu\nu$ -polynomials. If \mathcal{D} is an object of $\mu\nu\text{Poly}_{\mathcal{C}}$ and

$$H : \mathcal{D} \times \mathcal{C} \rightarrow \mathcal{C}$$

is a functor in $\mu\nu\text{Poly}_{\mathcal{C}}$, then $\mu H : \mathcal{D} \rightarrow \mathcal{C}$ and $\nu H : \mathcal{D} \rightarrow \mathcal{C}$ exist (and, hence, they are morphisms of $\mu\nu\text{Poly}_{\mathcal{C}}$).

Proof. Let X be any object of \mathcal{D} . Denoting by $X : \mathbb{1} \rightarrow \mathcal{D}$ the functor constantly equal to X , the functor H^X is the composition below.

$$\mathcal{C} \xrightarrow{(1, \text{id}_{\mathcal{C}})} \mathbb{1} \times \mathcal{C} \xrightarrow{(X \circ \pi_1, \text{id}_{\mathcal{C}} \circ \pi_2)} \mathcal{D} \times \mathcal{C} \xrightarrow{H} \mathcal{C}$$

$\underbrace{\hspace{15em}}_{H^X}$

Since all the morphisms above are in $\mu\nu\text{Poly}_{\mathcal{C}}$, we conclude that H^X is an endomorphism of $\mu\nu\text{Poly}_{\mathcal{C}}$. Therefore, since \mathcal{C} has $\mu\nu$ -polynomials, μH^X and νH^X exist.

By Proposition 4, since μH^X and νH^X exist for any X in \mathcal{D} , μH and νH exist. □

We say that a categorical semantics \mathcal{C} with (finite) sum and tuple types supports inductive and coinductive types if \mathcal{C} has chosen $\mu\nu$ -polynomials. Note that we do not consider the more general notion of (co)inductive types defined by endofunctors that may contain function types in their construction.

4. Structure-Preserving Functors

In this paper, the definition of our AD macro, the definitions of the concrete semantics, and logical relations are all framed in terms of appropriate structure-preserving functors. This fact highlights the significance of the suitable notions of structure-preserving functors in our work.

A structure-preserving functor between bicartesian closed categories are, of course, bicartesian closed functors. We usually assume that those are strict, which means that the functors preserve the structure on the nose.

It remains to establish the notion of structure-preserving functor between categories with $\mu\nu$ -polynomials. We do it below, starting by establishing the notion of preservation/creation/reflection of initial algebras and terminal coalgebras.

4.1 Preservation, reflection, and creation of initial algebras

We begin by recalling a fundamental result on lifting functors from the base categories to the categories of algebras in Lemma 9. This is actually related to the universal property of the categories of algebras.

Lemma 9. *Let $F : \mathcal{D} \rightarrow \mathcal{C}$ be a functor. Given endofunctors $E : \mathcal{C} \rightarrow \mathcal{C}$, $E' : \mathcal{D} \rightarrow \mathcal{D}$ and a natural transformation $\gamma : E \circ F \rightarrow F \circ E'$, we have an induced functor defined by:*

$$\begin{aligned} \tilde{F}_{\gamma} : E' \text{-Alg} &\rightarrow E \text{-Alg} \\ (X, \zeta) &\mapsto (F(X), F(\zeta) \circ \gamma_X) \\ g &\mapsto F(g). \end{aligned}$$

Proof. Indeed, if $g : W \rightarrow Z$ is the underlying morphism of an algebra morphism between (W, ζ) and (Z, ξ) , we have that

$$\begin{aligned} &F(g) \circ F(\zeta) \circ \gamma_W && \{ f : (W, \zeta) \rightarrow (Z, \xi) \} \\ &= F(\xi) \circ FE'(g) \circ \gamma_W && \{ \text{naturality of } \gamma \} \\ &= F(\xi) \circ \gamma_Z \circ EF(g) \end{aligned}$$

which proves that $F(g)$ in fact gives a morphism between the algebras $(F(W), F(\zeta) \circ \gamma_W)$ and $(F(Z), F(\xi) \circ \gamma_Z)$. The functoriality of \check{F}_γ follows, then, from that of F . \square

Dually, we have:

Lemma 10. *Let $E : \mathcal{C} \rightarrow \mathcal{C}$, $G : \mathcal{C} \rightarrow \mathcal{D}$, and $E' : \mathcal{D} \rightarrow \mathcal{D}$ be functors. Each natural transformation $\beta : G \circ E \rightarrow E' \circ G$ induces a functor:*

$$\begin{aligned} \tilde{G}^\beta : E\text{-CoAlg} &\rightarrow E'\text{-CoAlg} \\ (W, \xi) &\mapsto (G(W), \beta_W \circ G(\xi)) \\ f &\mapsto G(f). \end{aligned}$$

Below, whenever we talk about strict preservation, we are assuming that we have chosen initial objects (terminal objects) in the respective categories of (co)algebras.

We can, now, establish the definition of preservation, reflection, and creation of initial algebras using the respective notions for the induced functor. More precisely:

Definition 11 (Preservation, reflection, and creation of initial algebras). *We say that a functor $F : \mathcal{D} \rightarrow \mathcal{C}$ (strictly) preserves the initial algebra/reflects the initial algebra/creates the initial algebra of the endofunctor $E : \mathcal{C} \rightarrow \mathcal{C}$ if, whenever $E' : \mathcal{D} \rightarrow \mathcal{D}$ is such that $\gamma : E \circ F \cong F \circ E'$ (or, in the strict case, $F \circ E' = E \circ F$), the functor:*

$$\begin{aligned} \check{F}_\gamma : E'\text{-Alg} &\rightarrow E\text{-Alg} \\ (X, \zeta) &\mapsto (F(X), F(\zeta) \circ \gamma_X) \\ g &\mapsto F(g). \end{aligned}$$

induced by γ strictly) preserves the initial object/reflects the initial object/creates the initial object.

Finally, we say that a functor $F : \mathcal{D} \rightarrow \mathcal{C}$ (strictly) preserves initial algebras/reflects initial algebras/creates initial algebras if F (strictly) preserves initial algebras/reflects initial algebras/creates initial algebras of any endofunctor on \mathcal{D} .

Remark 12. In other words, let $F : \mathcal{D} \rightarrow \mathcal{C}$ be a functor.

- (I) We say that F (strictly) preserves initial algebras, if: for any natural isomorphism $\gamma : E \circ F \cong F \circ E'$ (or, in the strict case, for each identity $E \circ F = F \circ E'$) in which E and E' are endofunctors, assuming that $(\mu_{E'}, \text{in}_{E'})$ is the initial E' -algebra, the E -algebra $(F(\mu_{E'}), F(\text{in}_{E'}) \circ \gamma_{\mu_{E'}})$ is an initial object of $E\text{-Alg}$ (the chosen initial object of $E\text{-Alg}$, in the strict case).
- (II) We say that F reflects initial algebras, if: for any natural isomorphism $\gamma : E \circ F \cong F \circ E'$ in which E and E' are endofunctors, if $(F(Y), F(\xi) \circ \gamma_Y)$ is an initial E -algebra and (Y, ξ) is an E' -algebra, then (Y, ξ) is an initial E' -algebra.
- (III) We say that F creates initial algebras if: (A) F reflects and preserves initial algebras and, moreover, (B) for any $\gamma : E \circ F \cong F \circ E'$ in which E and E' are endofunctors, $E'\text{-Alg}$ has an initial algebra if $E\text{-Alg}$ does.

Definition 13 (Preservation, reflection, and creation of terminal coalgebras). *We say that a functor $G : \mathcal{C} \rightarrow \mathcal{D}$ (strictly) preserves the initial algebra/reflects the initial algebra/creates the initial algebra of an endofunctor $E : \mathcal{C} \rightarrow \mathcal{C}$ if, for any natural isomorphism $\beta : G \circ E \cong E' \circ G$ (or, in the strict case, $GE = E'G$), the functor:*

$$\tilde{G}^\beta : E\text{-CoAlg} \rightarrow E'\text{-CoAlg}$$

$$\begin{aligned} (W, \xi) &\mapsto (G(W), \beta_W \circ G(\xi)) \\ f &\mapsto G(f). \end{aligned}$$

induced by β (strictly) preserves the terminal object/reflects the terminal object/creates the terminal object.

Finally, we say that $G : \mathcal{C} \rightarrow \mathcal{D}$ (strictly) preserves terminal coalgebras/reflects terminal coalgebras/creates terminal coalgebras if G (strictly) preserves terminal coalgebras/reflects terminal coalgebras/creates terminal coalgebras of any endofunctor on \mathcal{C} .

4.2 $\mu\nu$ -polynomial-preserving functors

Finally, we can introduce the concept of a structure-preserving functor for $\mu\nu$ -polynomials.

Definition 14. A functor $G : \mathcal{D} \rightarrow \mathcal{C}$ (strictly) preserves $\mu\nu$ -polynomials if it strictly preserves finite coproducts, finite products, as well as initial algebras and terminal coalgebras of $\mu\nu$ -polynomials.

5. An Expressive Functional Language as a Source Language for AD

We describe a source language for our AD code transformations. We consider a standard total functional programming language with an expressive type system, over ground types \mathbf{real}^n for arrays of real numbers of static length n , for all $n \in \mathbb{N}$, and sets $\mathbf{Op}_{n_1, \dots, n_k}^m$ of primitive operations op , for all $k, m, n_1, \dots, n_k \in \mathbb{N}$. These operations op will be interpreted as differentiable functions $(\mathbb{R}^{n_1} \times \dots \times \mathbb{R}^{n_k}) \rightarrow \mathbb{R}^m$, and the reader can keep the following examples in mind:

- constants $\underline{c} \in \mathbf{Op}^n$ for each $c \in \mathbb{R}^n$, for which we slightly abuse notation and write $\underline{c}(\cdot)$ as \underline{c} ;
- elementwise addition and product $(+), (*) \in \mathbf{Op}_{n,n}^n$ and matrix-vector product $(\star) \in \mathbf{Op}_{n \cdot m, m}^n$;
- operations for summing all the elements in an array: $\text{sum} \in \mathbf{Op}_n^1$;
- some nonlinear functions like the sigmoid function $\zeta \in \mathbf{Op}_1^1$.

Its kinds, types, and terms are generated by the grammar in Fig. 1. We write $\Delta \vdash \tau : \text{type}$ to specify that the type τ is *well kinded* in *kinding context* Δ , where Δ is a list of the form $\alpha_1 : \text{type}, \dots, \alpha_n : \text{type}$. The idea is that the type variables identifiers $\alpha_1, \dots, \alpha_n$ can be used in the formation of τ . These kinding judgments are defined according to the rules displayed in Fig. 2. We write $\Delta \mid \Gamma \vdash t : \tau$ to specify that the term t is *well typed* in the *typing context* Γ , where Γ is a list of the form $x_1 : \tau_1, \dots, x_n : \tau_n$ for variable identifiers x_i and types τ_i that are well kinded in kinding context Δ . These typing judgments are defined according to the rules displayed in Fig. 3. As Fig. 4 displays, we consider the terms of our language up to the standard $\beta\eta$ -theory. To present this equational theory, we define in Fig. 5, by induction, some syntactic sugar for the functorial action $\Delta, \Delta' \mid \Gamma, x : \tau [\sigma/\alpha] \vdash \tau [x^t/\alpha] : \tau [\rho/\alpha]$ in argument α of parameterized types $\Delta, \alpha : \text{type} \vdash \tau : \text{type}$ on terms $\Delta' \mid \Gamma, x : \sigma \vdash t : \rho$.

We employ the usual conventions of free and bound variables and write $\tau [\sigma/\alpha]$ for the capture-avoiding substitution of the type σ for the identifier α in τ (and similarly, $t [s/x]$ for the capture-avoiding substitution of the term s for the identifier x in t). We define make liberal use of the standard syntactic sugar $\mathbf{let} \langle x, y \rangle = t \mathbf{in} s \stackrel{\text{def}}{=} \mathbf{let} z = t \mathbf{in} \mathbf{let} x = \mathbf{fst} z \mathbf{in} \mathbf{let} y = \mathbf{snd} z \mathbf{in} s$.

This standard language is equivalent to the freely generated bicartesian closed category \mathbf{Syn} with $\mu\nu$ -polynomials on the directed polygraph (computad) given by the ground types \mathbf{real}^n as objects and primitive operations op as arrows. Equivalently, we can see it as the initial category that supports tuple types, function types, sum types, inductive and coinductive types, and primitive types $\mathbf{Ty} = \{ \mathbf{real}^n \mid n \in \mathbb{N} \}$ and primitive operations $\mathbf{Op}(\mathbf{real}^{n_1}, \dots, \mathbf{real}^{n_k}; \mathbf{real}^m) = \mathbf{Op}_{n_1, \dots, n_k}^m$ (in the sense of Section 3). \mathbf{Syn} effectively represents programs as (categorical) combinators,

$\kappa, \kappa', \kappa'' ::=$ type	kinds kind of types
$\tau, \sigma, \rho ::=$ α real ⁿ 1 $\tau * \sigma$ $\tau \rightarrow \sigma$ $\{\ell_1 \tau_1 \mid \dots \mid \ell_n \tau_n\}$ $\mu \alpha. \tau$ $\nu \alpha. \tau$	(Cartesian) types type variable real arrays nullary product binary product function variant inductive type coinductive type
$t, s, r ::=$ x let $x = t$ in s $\text{op}(t_1, \dots, t_k)$ $\langle \rangle$ $\langle t, s \rangle$ fst t snd t $\lambda x. t$ $t s$ ℓt case t of $\{\ell_1 x_1 \rightarrow s_1 \mid \dots \mid \ell_n x_n \rightarrow s_n\}$ roll t fold t with $x \rightarrow s$ gen from t with $x \rightarrow s$ unroll t	terms variable let-bindings k -ary operations product tuples product projections function abstraction function application variant constructor variant match inductive constructor inductive destructor coinductive constructor coinductive destructor

Figure 1. Grammar for the kinds, types, and terms of the source language for our AD transformations.

$$\begin{array}{c}
 \frac{((\alpha : \text{type}) \in \Delta)}{\Delta \vdash \alpha : \text{type}} \quad \frac{}{\Delta \vdash \mathbf{real}^n : \text{type}} \quad \frac{}{\Delta \vdash \mathbf{1} : \text{type}} \quad \frac{\Delta \vdash \tau : \text{type} \quad \Delta \vdash \sigma : \text{type}}{\Delta \vdash \tau * \sigma : \text{type}} \\
 \\
 \frac{\cdot \vdash \tau : \text{type} \quad \cdot \vdash \sigma : \text{type}}{\cdot \vdash \tau \rightarrow \sigma : \text{type}} \quad \frac{\{\Delta \vdash \tau_i : \text{type} \quad \ell_i \text{ label}\}_{1 \leq i \leq n}}{\Delta \vdash \{\ell_1 \tau_1 \mid \dots \mid \ell_n \tau_n\} : \text{type}} \\
 \\
 \frac{\Delta, \alpha : \kappa \vdash \tau : \text{type}}{\Delta \vdash \mu \alpha. \tau : \text{type}} \quad \frac{\Delta, \alpha : \kappa \vdash \tau : \text{type}}{\Delta \vdash \nu \alpha. \tau : \text{type}}
 \end{array}$$

Figure 2. Kinding rules for the AD source language. Note that we only consider the formation of function types of nonparameterized types (shaded in gray).

also known as “point-free style” in the functional programming community. Concretely, **Syn** has types as objects, homsets **Syn**(τ, σ) consist of $(\alpha)\beta\eta$ -equivalence classes of terms $\cdot \mid x : \tau \vdash t : \sigma$, identities are $\cdot \mid x : \tau \vdash x : \tau$, and the composition of $\cdot \mid x : \tau \vdash t : \sigma$ and $\cdot \mid y : \sigma \vdash s : \rho$ is given by $\cdot \mid x : \tau \vdash \mathbf{let} \ y = t \ \mathbf{in} \ s : \rho$.

$$\begin{array}{c}
 \frac{((x : \tau) \in \Gamma)}{\Delta | \Gamma \vdash x : \tau} \quad \frac{\Delta | \Gamma \vdash t : \tau \quad \Delta | \Gamma, x : \tau \vdash s : \sigma}{\Delta | \Gamma \vdash \mathbf{let} x = t \mathbf{ in } s : \sigma} \quad \frac{\{\Delta | \Gamma \vdash t_i : \mathbf{real}^{n_i}\}_{i=1}^k \quad (\text{op} \in \text{Op}_{n_1, \dots, n_k}^m)}{\Delta | \Gamma \vdash \text{op}(t_1, \dots, t_k) : \mathbf{real}^m} \\
 \\
 \frac{}{\Delta | \Gamma \vdash \langle \rangle : \mathbf{1}} \quad \frac{\Delta | \Gamma \vdash t : \tau \quad \Delta | \Gamma \vdash s : \sigma}{\Delta | \Gamma \vdash \langle t, s \rangle : \tau * \sigma} \quad \frac{\Delta | \Gamma \vdash t : \tau * \sigma}{\Delta | \Gamma \vdash \mathbf{fst} t : \tau} \quad \frac{\Delta | \Gamma \vdash t : \tau * \sigma}{\Delta | \Gamma \vdash \mathbf{snd} t : \sigma} \\
 \\
 \frac{\Delta | \Gamma, x : \tau \vdash t : \sigma}{\Delta | \Gamma \vdash \lambda x. t : \tau \rightarrow \sigma} \quad \frac{\Delta | \Gamma \vdash t : \sigma \rightarrow \tau \quad \Delta | \Gamma \vdash s : \sigma}{\Delta | \Gamma \vdash t s : \tau} \\
 \\
 \frac{\Delta | \Gamma \vdash t : \tau_i}{\Delta | \Gamma \vdash \ell_i t : \{\ell_1 \tau_1 | \dots | \ell_n \tau_n\}} \quad \frac{\Delta | \Gamma \vdash t : \{\ell_1 \tau_1 | \dots | \ell_n \tau_n\} \quad \{\Delta | \Gamma, x_i : \tau_i \vdash s_i : \rho\}_{1 \leq i \leq n}}{\Delta | \Gamma \vdash \mathbf{case} t \mathbf{ of } \{\ell_1 x_1 \rightarrow s_1 | \dots | \ell_n x_n \rightarrow s_n\} : \rho} \\
 \\
 \frac{\Delta | \Gamma \vdash t : \tau[\mu\alpha.\tau/\alpha]}{\Delta | \Gamma \vdash \mathbf{roll} t : \mu\alpha.\tau} \quad \frac{\Delta | \Gamma \vdash t : \mu\alpha.\tau \quad \Delta | x : \tau[\sigma/\alpha] \vdash s : \sigma}{\Delta | \Gamma \vdash \mathbf{fold} t \mathbf{ with } x \rightarrow s : \sigma} \\
 \\
 \frac{\Delta | \Gamma \vdash t : \sigma \quad \Delta | x : \sigma \vdash s : \tau[\sigma/\alpha]}{\Delta | \Gamma \vdash \mathbf{gen from} t \mathbf{ with } x \rightarrow s : \nu\alpha.\tau} \quad \frac{\Delta | \Gamma \vdash t : \nu\alpha.\tau}{\Delta | \Gamma \vdash \mathbf{unroll} t : \tau[\nu\alpha.\tau/\alpha]}
 \end{array}$$

Figure 3. Typing rules for the AD source language.

$$\begin{array}{l}
 \mathbf{let} x = t \mathbf{ in } s = s[t/x] \quad t = \langle \rangle \quad \mathbf{fst} \langle t, s \rangle = t \quad \mathbf{snd} \langle t, s \rangle = s \quad t = \langle \mathbf{fst} t, \mathbf{snd} t \rangle \\
 \\
 (\lambda x. t) s = t[s/x] \quad t \stackrel{\#x}{=} \lambda x. t x \quad \mathbf{case} \ell_i t \mathbf{ of } \{\ell_1 x_1 \rightarrow s_1 | \dots | \ell_n x_n \rightarrow s_n\} = s_i[t/x_i] \\
 \\
 s[t/y] \stackrel{\#x_1, \dots, x_n}{=} \mathbf{case} t \mathbf{ of } \{\ell_1 x_1 \rightarrow s[\ell_1 x_1/y] | \dots | \ell_n x_n \rightarrow s[\ell_n x_n/y]\} \\
 \\
 \mathbf{fold roll} t \mathbf{ with } x \rightarrow s \stackrel{\#y}{=} s[\tau[\nu\alpha.\mathbf{fold} y \mathbf{ with } x \rightarrow s/\alpha][t/y]/x] \\
 \\
 r[\mathbf{roll} x/z] = s[\tau[x\tau/\alpha]/z] \text{ implies } r[t/x] = \mathbf{fold} t \mathbf{ with } z \rightarrow s \\
 \\
 \mathbf{unroll} (\mathbf{gen from} t \mathbf{ with } x \rightarrow s) \stackrel{\#y}{=} \tau[\nu\alpha.\mathbf{gen from} y \mathbf{ with } x \rightarrow s/\alpha][s/y, t/x] \\
 \\
 \mathbf{unroll} r = \tau[x\tau/\alpha][s/x] \text{ implies } r[t/x] = \mathbf{gen from} t \mathbf{ with } x \rightarrow s
 \end{array}$$

Figure 4. We consider the standard $\beta\eta$ -laws above for our language. We write $\stackrel{\#x_1, \dots, x_n}{=}$ to indicate that the variables x_1, \dots, x_n need to be fresh in the left-hand side. Equations hold on pairs of terms of the same type. As usual, we only distinguish terms up to α -renaming of bound variables.

Corollary 15 (Universal property of **Syn**). *Given any bicartesian closed category with $\mu\nu$ -polynomials \mathcal{C} , any consistent assignment of $F(\mathbf{real}^n) \in \text{obj}(\mathcal{C})$ and $F(\text{op}) \in \mathcal{C}(F(\mathbf{real}^{n_1}) \times \dots \times F(\mathbf{real}^{n_k}), F(\mathbf{real}^m))$ for $\text{op} \in \text{Op}_{n_1, \dots, n_k}^m$ extends to a unique $\mu\nu$ -polynomial-preserving bicartesian closed functor $F : \mathbf{Syn} \rightarrow \mathcal{C}$.*

6. Modeling Expressive Functional Languages in Grothendieck Constructions

In this section, we present a novel construction of categorical models (in the sense of Section 3) $\Sigma_{\mathcal{C}}\mathcal{L}$ and $\Sigma_{\mathcal{C}}\mathcal{L}^{op}$ of expressive functional languages (like our AD source language of Section 5) in Σ -types of suitable indexed categories $\mathcal{L} : \mathcal{C}^{op} \rightarrow \mathbf{Cat}$. In particular, the problem we solve in this section is to identify suitable sufficient conditions to put on an indexed category $\mathcal{L} : \mathcal{C}^{op} \rightarrow \mathbf{Cat}$, whose base category we think of as the semantics of a cartesian type theory and whose fiber

$$\begin{aligned}
 \alpha[x^{\tau}/\alpha] &= t \\
 \beta[x^{\tau}/\alpha] &= x \quad \text{if } \alpha \neq \beta \\
 \mathbf{real}^n[x^{\tau}/\alpha] &= x \\
 \mathbf{1}[x^{\tau}/\alpha] &= x \\
 (\tau * \sigma)[x^{\tau}/\alpha] &= \langle \tau[x^{\tau}/\alpha][\mathbf{fst} \ x/x], \sigma[x^{\tau}/\alpha][\mathbf{snd} \ x/x] \rangle \\
 \{\ell_1 \tau_1 \mid \dots \mid \ell_n \tau_n\}[x^{\tau}/\alpha] &= \mathbf{case} \ x \ \mathbf{of} \ \{\ell_1 x \rightarrow \ell_1 \tau_1[x^{\tau \ell_1 t}/\alpha] \mid \dots \mid \ell_n x \rightarrow \ell_n \tau_n[x^{\tau \ell_n t}/\alpha]\} \\
 (\mu \alpha. \tau)[x^{\tau}/\alpha] &= x \\
 (\mu \beta. \tau)[x^{\tau}/\alpha] &= \mathbf{fold} \ x \ \mathbf{with} \ x \rightarrow \mathbf{roll} \ \tau[x^{\tau}/\alpha] \quad \text{if } \alpha \neq \beta \\
 (\nu \alpha. \tau)[x^{\tau}/\alpha] &= x \\
 (\nu \beta. \tau)[x^{\tau}/\alpha] &= \mathbf{gen} \ \mathbf{from} \ x \ \mathbf{with} \ x \rightarrow \tau[x^{\tau}/\alpha][\mathbf{unroll} \ x/x] \quad \text{if } \alpha \neq \beta
 \end{aligned}$$

Figure 5. Functorial action $\Delta, \Delta' \mid \Gamma, x : \tau[\sigma/\alpha] \vdash \tau[x^{\tau}/\alpha] : \tau[\rho/\alpha]$ in argument α of parameterized types $\Delta, \alpha : \text{type} \vdash \tau : \text{type}$ on terms $\Delta' \mid \Gamma, x : \sigma \vdash t : \rho$ of the source language.

categories we think of as the semantics of a dependent linear type theory, such that $\Sigma_C \mathcal{L}$ and $\Sigma_C \mathcal{L}^{op}$ are categorical models of expressive functional languages in this sense. We call such an indexed category a Σ -bimodel of language feature X if it satisfies our sufficient conditions for $\Sigma_C \mathcal{L}$ and $\Sigma_C \mathcal{L}^{op}$ to be categorical models of language feature X .

This abstract material in many ways forms the theoretical crux of this paper. We consider two particular instances of this idea later:

- the case where \mathcal{L} is the syntactic category $\mathbf{LSyn} : \mathbf{CSyn}^{op} \rightarrow \mathbf{Cat}$ of a suitable target language for AD translations (Section 7); the universal property of the source language \mathbf{Syn} then yields unique structure-preserving functors $\overrightarrow{\mathcal{D}} : \mathbf{Syn} \rightarrow \Sigma_{\mathbf{CSyn}} \mathbf{LSyn}$ and $\overleftarrow{\mathcal{D}} : \mathbf{Syn} \rightarrow \Sigma_{\mathbf{CSyn}} \mathbf{LSyn}^{op}$ implementing forward and reverse-mode AD;
- the case where \mathcal{L} is the indexed category of families of real vector spaces $\mathbf{FVect} : \mathbf{Set}^{op} \rightarrow \mathbf{Cat}$ (Section 9); this gives a concrete denotational semantics to the target language, which we use in the correctness proof of AD.

6.1 Basics: the categories $\Sigma_C \mathcal{L}$ and $\Sigma_C \mathcal{L}^{op}$

Recall that for any strictly indexed category, that is, a (strict) functor $\mathcal{L} : \mathcal{C}^{op} \rightarrow \mathbf{Cat}$, we can consider its total category (or Grothendieck construction) $\Sigma_C \mathcal{L}$, which is a fibered category over \mathcal{C} (see Johnstone 2002, Sections A1.1.7, B1.3.1). We can view it as a Σ -type of categories, which generalizes the cartesian product. Further, given a strictly indexed category $\mathcal{L} : \mathcal{C}^{op} \rightarrow \mathbf{Cat}$, we can consider its fiberwise dual category $\mathcal{L}^{op} : \mathcal{C}^{op} \rightarrow \mathbf{Cat}$, which is defined as the composition $\mathcal{C}^{op} \xrightarrow{\mathcal{L}} \mathbf{Cat} \xrightarrow{op} \mathbf{Cat}$, where op is defined by $A \mapsto A^{op}$. Thus, we can apply the same construction to \mathcal{L}^{op} to obtain a category $\Sigma_C \mathcal{L}^{op}$.

Concretely, $\Sigma_C \mathcal{L}$ is the following category:

- objects are pairs (W, w) of an object W of \mathcal{C} and an object w of $\mathcal{L}(W)$;
- morphisms $(W, w) \rightarrow (X, x)$ are pairs (f, f') with $f : W \rightarrow X$ in \mathcal{C} and $f' : w \rightarrow \mathcal{L}(f)(x)$ in $\mathcal{L}(W)$;
- identities $\text{id}_{(W, w)}$ are $(\text{id}_W, \text{id}_w)$;
- composition of $(W, w) \xrightarrow{(f, f')} (X, x)$ and $(X, x) \xrightarrow{(g, g')} (Y, y)$ is given by:

$$(g \circ f, \mathcal{L}(f)(g') \circ f').$$

Concretely, $\Sigma_C \mathcal{L}^{op}$ is the following category:

- objects are pairs (W, w) of an object W of \mathcal{C} and an object w of $\mathcal{L}(W)$;
- morphisms $(W, w) \rightarrow (X, x)$ are pairs (f, f') with $f : W \rightarrow X$ in \mathcal{C} and $f' : \mathcal{L}(f)(x) \rightarrow w$ in $\mathcal{L}(W)$;
- identities $\text{id}_{(W,w)}$ are $(\text{id}_W, \text{id}_W)$;
- composition of $(W, w) \xrightarrow{(f,f')} (X, x)$ and $(X, x) \xrightarrow{(g,g')} (Y, y)$ is given by:

$$(g \circ f, f' \circ \mathcal{L}(f)(g')).$$

6.2 Products in total categories

We start by studying the cartesian structure of $\Sigma_{\mathcal{C}}\mathcal{L}$. We refer to Gray (1966) for a basic reference for fibrations/indexed categories and properties of the total category.

Definition 16. A strictly indexed category \mathcal{L} has strictly indexed finite (co)products if

- (i) each fiber $\mathcal{L}(C)$ has chosen finite (co)products $(\times, \mathbb{1})$ (respectively, $(\sqcup, 0)$);
- (ii) change of base strictly preserves these (co)products in the sense that $\mathcal{L}(f)$ preserves finite products (respectively, finite coproducts) for all morphisms f in \mathcal{C} .

We recall the well-known fact that $\Sigma_{\mathcal{C}}\mathcal{L}$ ($\Sigma_{\mathcal{C}}\mathcal{L}^{\text{op}}$) has finite products if \mathcal{C} has finite products and \mathcal{L} has indexed finite products (coproducts).

Proposition 17 (Cartesian structure of $\Sigma_{\mathcal{C}}\mathcal{L}$). Assuming that \mathcal{C} has finite products $(\mathbb{1}, \times)$ and \mathcal{L} has indexed finite products $(\mathbb{1}, \times)$, we have that $\Sigma_{\mathcal{C}}\mathcal{L}$ has (fibered) terminal object $\mathbb{1} = (\mathbb{1}, \mathbb{1})$ and (fibered) binary product $(W, w) \times (Y, y) = (W \times Y, \mathcal{L}(\pi_1)(w) \times \mathcal{L}(\pi_2)(y))$.

Proof. We have (natural) bijections:

$$\begin{aligned} & \Sigma_{\mathcal{C}}\mathcal{L}((X, x), (\mathbb{1}, \mathbb{1})) \\ &= \Sigma_{f \in \mathcal{C}(X, \mathbb{1})} \mathcal{L}(X)(x, \mathcal{L}(f)(\mathbb{1})) && \{ \text{by definition} \} \\ &\cong \Sigma_{f \in \mathcal{C}(X, \mathbb{1})} \mathcal{L}(X)(x, \mathbb{1}) && \{ \text{indexed } \mathbb{1} \} \\ &\cong \mathbb{1} \times \mathbb{1} && \{ \mathbb{1} \text{ terminal in } \mathcal{C} \text{ and } \mathcal{L}(X) \} \\ &\cong \mathbb{1} \end{aligned}$$

$$\begin{aligned} & \Sigma_{\mathcal{C}}\mathcal{L}((X, x), (W \times Z, \mathcal{L}(\pi_1)(w) \times \mathcal{L}(\pi_2)(z))) \\ &= \Sigma_{(f,g) \in \mathcal{C}(X, W \times Z)} \mathcal{L}(X)(x, \mathcal{L}(f, g)(\mathcal{L}(\pi_1)(w) \times \mathcal{L}(\pi_2)(z))) && \{ \text{by definition} \} \\ &\cong \Sigma_{(f,g) \in \mathcal{C}(X, W \times Z)} \mathcal{L}(X)(x, \mathcal{L}(f, g)\mathcal{L}(\pi_1)(w) \times \mathcal{L}(f, g)\mathcal{L}(\pi_2)(z)) && \{ \text{indexed } \times \} \\ &= \Sigma_{(f,g) \in \mathcal{C}(X, W \times Z)} \mathcal{L}(X)(x, \mathcal{L}(f)(w) \times \mathcal{L}(g)(z)) && \{ \text{functoriality } \mathcal{L} \} \\ &\cong \Sigma_{(f,g) \in \mathcal{C}(X, W \times Z)} \mathcal{L}(X)(x, \mathcal{L}(f)(w)) \times \mathcal{L}(X)(x, \mathcal{L}(g)(z)) && \{ \times \text{ product in } \mathcal{L}(A_1) \} \\ &\cong \Sigma_{f \in \mathcal{C}(X, W)} \Sigma_{g \in \mathcal{C}(X, Z)} \mathcal{L}(X)(x, \mathcal{L}(f)(w)) \times \mathcal{L}(X)(x, \mathcal{L}(g)(z)) && \{ \times \text{ product in } \mathcal{C} \} \\ &\cong (\Sigma_{f \in \mathcal{C}(X, W)} \mathcal{L}(X)(x, \mathcal{L}(f)(w))) \times (\Sigma_{g \in \mathcal{C}(X, Z)} \mathcal{L}(X)(x, \mathcal{L}(g)(z))) && \{ \text{Beck-Chevalley for } \Sigma \text{ in Set} \} \\ &= \Sigma_{\mathcal{C}}\mathcal{L}((X, x), (W, w)) \times \Sigma_{\mathcal{C}}\mathcal{L}((X, x), (Z, z)). \end{aligned}$$

□

In particular, finite products in $\Sigma_{\mathcal{C}}\mathcal{L}$ are fibered in the sense that the projection functor $\Sigma_{\mathcal{C}}\mathcal{L} \rightarrow \mathcal{C}$ preserves them, on the nose. Codually, we have:

Proposition 18 (Cartesian structure of $\Sigma_{\mathcal{C}}\mathcal{L}^{\text{op}}$). Assuming that \mathcal{C} has finite products $(\mathbb{1}, \times)$ and \mathcal{L} has indexed finite coproducts $(0, \sqcup)$, we have that $\Sigma_{\mathcal{C}}\mathcal{L}^{\text{op}}$ has (fibered) terminal object $\mathbb{1} = (\mathbb{1}, 0)$ and (fibered) binary product $(W, w) \times (Y, y) = (W \times Y, \mathcal{L}(\pi_1)(w) \sqcup \mathcal{L}(\pi_2)(y))$.

That is, in our terminology, $\mathcal{L} : \mathcal{C}^{op} \rightarrow \mathbf{Cat}$ is a Σ -bimodel of tuple types if \mathcal{C} has chosen finite products and \mathcal{L} has finite strictly indexed products and coproducts.

We will, in particular, apply the results above in the situation where \mathcal{L} has indexed finite biproducts in the sense of Definition 19, in which case the finite product structures of $\Sigma_{\mathcal{C}}\mathcal{L}$ and $\Sigma_{\mathcal{C}}\mathcal{L}^{op}$ coincide.

Definition 19 (Strictly indexed finite biproducts). *A category with finite products and coproducts is semi-additive if the binary coproduct functor is naturally isomorphic to the binary product functor; see, for instance, Lack (2012), Lucatelli Nunes (2019). In this case, the product/coproduct is called biproduct, and the biproduct structure is denoted by $(\times, \mathbb{1})$ or $(+, \mathbb{0})$.*

A strictly indexed category \mathcal{L} has strictly indexed finite biproducts if

- \mathcal{L} has strictly indexed finite products and coproducts;
- each fiber $\mathcal{L}(C)$ is semi-additive.

6.3 Generators

In this section, we establish the obvious sufficient (and necessary) conditions for $\Sigma_{\mathcal{C}}\mathcal{L}$ and $\Sigma_{\mathcal{C}}\mathcal{L}^{op}$ to model primitive types and operations in the sense of Section 3. These conditions are an immediate consequence of the structure of $\Sigma_{\mathcal{C}}\mathcal{L}$ and $\Sigma_{\mathcal{C}}\mathcal{L}^{op}$ as cartesian categories.

Definition 20. *We say that $\mathcal{L} : \mathcal{C}^{op} \rightarrow \mathbf{Cat}$ is a Σ -bimodel of primitive types \mathbf{Ty} and operations \mathbf{Op} if*

- for all $T \in \mathbf{Ty}$, we have a choice of objects $C_T \in \text{obj}(\mathcal{C})$ and $L_T, L'_T \in \text{obj}(\mathcal{L})(C_T)$;
- for all $\text{op} \in \mathbf{Op}(T_1, \dots, T_n; S)$, we have a choice of morphisms:

$$\begin{aligned} f_{\text{op}} &\in \mathcal{C}(C_{T_1} \times \dots \times C_{T_n}, C_S) \\ g_{\text{op}} &\in \mathcal{L}(C_{T_1} \times \dots \times C_{T_n})(\mathcal{L}(\pi_1)(L_{T_1}) \times \dots \times \mathcal{L}(\pi_n)(L_{T_n}), \mathcal{L}(f_{\text{op}})(L_S)) \\ g'_{\text{op}} &\in \mathcal{L}(C_{T_1} \times \dots \times C_{T_n})(\mathcal{L}(f_{\text{op}})(L'_S), \mathcal{L}(\pi_1)(L'_{T_1}) \sqcup \dots \sqcup \mathcal{L}(\pi_n)(L'_{T_n})). \end{aligned}$$

We say that such a model has self-dual primitive types in case $L_T = L'_T$ for all $T \in \mathbf{Ty}$.

6.4 Cartesian closedness of total categories

The question of Cartesian closure of the categories $\Sigma_{\mathcal{C}}\mathcal{L}$ and $\Sigma_{\mathcal{C}}\mathcal{L}^{op}$ is a lot more subtle. In particular, the formulas for exponentials tend to involve Π - and Σ -types; hence, we need to recall some definitions from categorical dependent type theory. As also suggested by Kerjean and Pédrot (2021), these formulas relate closely to the Diller–Nahm variant (Diller 1974; Hyland 2002; Moss and von Glehn 2018) of the Dialectica interpretation (Gödel 1958) and Altenkirch et al. (2010)’s formula for higher-order containers. We plan to explain this connection in detail in future work as it would form a distraction from the point of the current paper.

We use standard definitions from the semantics of dependent type theory and the dependently typed enriched effect calculus. An interested reader can find background on this material in Vákár (2017, Chapter 5) and Ahman et al. (2016). We briefly recalling some of the usual vocabulary (Vákár 2017, Chapter 5).

Definition 21. *Given an indexed category $\mathcal{D} : \mathcal{C}^{op} \rightarrow \mathbf{Cat}$, we say:*

- it satisfies the comprehension axiom if: \mathcal{C} has a chosen terminal object $\mathbb{1}$; \mathcal{D} has strictly indexed terminal objects $\mathbb{1}$ (i.e., chosen terminal objects $\mathbb{1} \in \mathcal{D}(X)$, such that $\mathcal{D}(g)(\mathbb{1}) = \mathbb{1} \in \mathcal{D}(W)$ for all $g : W \rightarrow X$ in \mathcal{C}); and, for each object $(X, x) \in \Sigma_{\mathcal{C}}\mathcal{D}$, the functor:

$$\begin{aligned} \mathbf{rc}_{(X,x)} : (\mathcal{C}/X)^{op} &\rightarrow \mathbf{Set} \\ \left(W, W \xrightarrow{f} X \right) &\mapsto \mathcal{D}(W)(\mathbb{1}, \mathcal{D}(f)(x)) \end{aligned}$$

are representable by an object $(X.x, X.x \xrightarrow{\mathbf{p}_{X,x}} X)$ of \mathcal{C}/X :

$$\begin{aligned} \mathbf{rc}_{(X,x)} \left(W, W \xrightarrow{f} X \right) &= \mathcal{D}(W)(\mathbb{1}, \mathcal{D}(f)(x)) \cong \mathcal{C}/X \left((W, f), (X.x, \mathbf{p}_{X,x}) \right) \\ b &\mapsto (f, g). \end{aligned}$$

We write $\mathbf{v}_{X,x}$ for the unique element of $\mathcal{D}(X.x)(\mathbb{1}, \mathcal{D}(\mathbf{p}_{X,x})(x))$ such that $(\mathbf{p}_{X,x}, \mathbf{v}_{X,x}) = \mathbf{id}_{\mathbf{p}_{X,x}}$ (the universal element of the representation).

Furthermore, given $f : W \rightarrow X$, we write $\mathbf{q}_{f,b}$ for the unique morphism $(f \circ \mathbf{p}_{W,\mathcal{D}(f)(x)}, \mathbf{v}_{W,\mathcal{D}(f)(x)})$ making the square below a pullback:

$$\begin{array}{ccc} W.\mathcal{D}(f)(x) & \xrightarrow{\mathbf{p}_{f,x}} & X.x \\ \mathbf{p}_{X,\mathcal{D}(f)(x)} \downarrow & & \downarrow \mathbf{p}_{X,x} \\ W & \xrightarrow{f} & X \end{array}$$

We henceforth call such squares **p**-squares;

- it supports Σ -types if we have left adjoint functors $\Sigma_w \dashv \mathcal{D}(\mathbf{p}_{W,w}) : \mathcal{D}(W.w) \rightleftarrows \mathcal{D}(W)$ satisfying the left Beck–Chevalley condition for **p**-squares w.r.t. \mathcal{D} (this means that $\mathcal{D}(f) \circ (\Sigma_{\mathcal{D}(f)(x)} \rightarrow \Sigma_x) \circ \mathcal{D}(\mathbf{p}_{f,x})$ are the identity);
- it supports Π -types if \mathcal{D}^{op} supports Σ -types; explicitly, that is the case iff we have right adjoint functors $\mathcal{D}(\mathbf{p}_{W,w}) \dashv \Pi_w : \mathcal{D}(W) \rightleftarrows \mathcal{D}(W.w)$ satisfying the right Beck–Chevalley condition for **p**-squares in the sense that the canonical maps $\Pi_{\mathcal{D}(f)(x)} \circ (\mathcal{D}(f) \rightarrow \mathcal{D}(\mathbf{p}_{f,x})) \circ \Pi_x$ are the identity.

Definition 22. In case $\mathcal{D} : \mathcal{C}^{op} \rightarrow \mathbf{Cat}$ satisfies the comprehension axiom, we say that

- it satisfies democratic comprehension if the comprehension functor:

$$\begin{aligned} \mathcal{D}(W)(w', w) &\xrightarrow{\mathbf{p}_{W,w'}} \mathcal{C}/W \left((W.w', \mathbf{p}_{W,w'}), (W.w, \mathbf{p}_{W,w}) \right) \\ d &\mapsto (\mathbf{p}_{W,w'}, \mathcal{D}(\mathbf{p}_{W,w'})(d) \circ \mathbf{v}_{W,w'}) \end{aligned}$$

defines an isomorphism of categories $\mathcal{D}(\mathbb{1}) \cong \mathcal{C}/\mathbb{1} \cong \mathcal{C}$;

- it satisfies full/faithful comprehension if the comprehension functor is full/faithful;
- it supports (strong) Σ -types (i.e., Σ -types with a dependent elimination rule, which in particular makes \mathcal{D} support Σ -types) if dependent projections compose: for all triple (W, w, s) where $W \in \mathcal{C}$, $w \in \text{obj}(\mathcal{D}(W))$ and $s \in \text{obj}(\mathcal{D}(W.w))$, we have

$$\mathbf{p}_{W,w} \circ \mathbf{p}_{W.w,s} \cong \mathbf{p}_{W,\Sigma_w s}$$

then, in particular, $W.\Sigma_w s \cong W.w.s$; further, we have projection morphisms $\pi_1 \in \mathcal{D}(W)(\Sigma_w s, w)$ and $\pi_2 \in \mathcal{D}(W.w)(\mathbb{1}, s)$;

Remark 23 (Σ - and Π - as dependent product and function types). In case, \mathcal{D} satisfies fully faithful comprehension,

- $\Sigma_w \mathcal{D}(\mathbf{p}_{W,w})(v)$ gives the categorical product $w \times v$ of w and v in $\mathcal{D}(W)$;
- $\Pi_w \mathcal{D}(\mathbf{p}_{W,w})(v)$ gives the categorical exponential $w \Rightarrow v$ of w and v in $\mathcal{D}(W)$.

Definition 24 (Σ -bimodel for function types). We call a strictly indexed category $\mathcal{L} : \mathcal{C}^{op} \rightarrow \mathbf{Cat}$ a Σ -bimodel for function types if it is a biadditive model of the dependently typed enriched effect calculus in the sense that it comes equipped with

- ($\mathcal{L}A$) a model of cartesian dependent type theory in the sense of a strictly indexed category $\mathcal{C}' : \mathcal{C}^{op} \rightarrow \mathbf{Cat}$ that satisfies full, faithful, democratic comprehension with Π -types and strong Σ -types;
- ($\mathcal{L}B$) strictly indexed finite biproducts in the sense of Definition 19 in \mathcal{L} ;
- ($\mathcal{L}C$) Σ - and Π -types in \mathcal{L} ;
- ($\mathcal{L}D$) a strictly indexed functor $\dashv : \mathcal{L}^{op} \times \mathcal{L} \rightarrow \mathcal{C}'$ and a natural isomorphism:

$$\mathcal{L}(W)(w, x) \cong \mathcal{C}'(A)(\mathbb{1}, w \dashv x).$$

We can immediately note that our notion of Σ -bimodel of function types is also a Σ -bimodel of tuple types. Indeed, strong Σ -types and comprehension give us, in particular, chosen finite products in \mathcal{C} .

We next show why this name is justified: we show that the Grothendieck construction of a Σ -bimodel of function types is cartesian closed.⁴

In the following, we slightly abuse notation to aid legibility:

- denoting by $!_W : W \rightarrow \mathbb{1}$ the only morphism, we will sometimes conflate $Z \in \text{obj} \mathcal{C}'(\mathbb{1})$ and $\mathbb{1}.Z \in \text{obj}(\mathcal{C})$ as well as $f \in \mathcal{C}'(W)(\mathbb{1}, \mathcal{C}'(!_W)(Z))$ and $(!_W, f) \in \mathcal{C}(W, \mathbb{1}.Z)$; this is justified by the democratic comprehension axiom;
- we will sometimes simply write z for $\mathcal{D}(\mathbf{p}_{W,w})(z)$ where the weakening map $\mathcal{D}(\mathbf{p}_{W,w})$ is clear from context.

Given $X, Y \in \mathcal{C}$ we will write ev_1 for the obvious \mathcal{C} -morphism

$$\text{ev}_1 : \Pi_X \Sigma_Y Z.X \rightarrow Y,$$

that is, the morphism obtained as the composition (where we write π_1 for the projection $\Sigma_Y Z \rightarrow Y$):

$$\Pi_X \Sigma_Y Z.X \cong (\Pi_X \Sigma_Y Z) \times X \xrightarrow{(\Pi_X \pi_1) \times X} (\Pi_X Y) \times X \cong (X \Rightarrow Y) \times X \xrightarrow{\text{ev}} Y$$

With these notational conventions in place, we can describe the cartesian closed structure of Grothendieck constructions.

Theorem 25 (Exponentials of the total category). For a Σ -bimodel \mathcal{L} for function types, $\Sigma_{\mathcal{C}} \mathcal{L}$ has exponential:

$$(X, x) \Rightarrow (Y, y) = (\Pi_X \Sigma_Y \mathcal{L}(\pi_1)(x) \dashv \mathcal{L}(\pi_2)(y), \Pi_X \mathcal{L}(\text{ev}_1)(y)).$$

Proof. We have (natural) bijections:

$$\begin{aligned} \Sigma_{\mathcal{C}} \mathcal{L}((W, w) \times (X, x), (Y, y)) &= \\ &= \Sigma_{\mathcal{C}} \mathcal{L}((W \times X, \mathcal{L}(\pi_1)(w) \times \mathcal{L}(\pi_2)(x)), (Y, y)) && \{ \text{by Prop. 17} \} \\ &= \Sigma_{f \in \mathcal{C}(W \times X, Y)} \mathcal{L}(W \times X)(\mathcal{L}(\pi_1)(w) \times \mathcal{L}(\pi_2)(x), \mathcal{L}(f)(y)) && \{ \text{by definition} \} \end{aligned}$$

$$\begin{aligned}
 &\cong \sum_{f \in \mathcal{C}(W \times X, Y)} \mathcal{L}(W \times X)(\mathcal{L}(\pi_1)(w), \mathcal{L}(f)(y)) \times \mathcal{L}(W \times X)(\mathcal{L}(\pi_2)(x), \mathcal{L}(f)(y)) && \{ \times \text{ coproduct in } \mathcal{L}(W \times X) \} \\
 &\cong \sum_{f \in \mathcal{C}(W \times X, Y)} \mathcal{L}(W)(w, \Pi_X \mathcal{L}(f)(y)) \times \mathcal{L}(W \times X)(\mathcal{L}(\pi_2)(x), \mathcal{L}(f)(y)) && \{ \Pi\text{-types in } \mathcal{L} \} \\
 &\cong \sum_{f \in \mathcal{C}(W \times X, Y)} \mathcal{L}(W)(w, \Pi_X \mathcal{L}(f)(y)) \times \mathcal{C}'(W \times X)(\mathbb{1}, \mathcal{L}(\pi_2)(x) \multimap \mathcal{L}(f)(y)) && \{ \multimap\text{-types in } \mathcal{C}' \} \\
 &\cong \sum_{(f, g) \in \sum_{f \in \mathcal{C}(W \times X, Y)} \mathcal{C}'(W \times X)(\mathbb{1}, \mathcal{L}(\pi_2)(x) \multimap \mathcal{L}(f)(y))} \mathcal{L}(W)(w, \Pi_X \mathcal{L}(f)(y)) && \{ \Sigma\text{-types in Set} \} \\
 &\cong \sum_{(f, g) \in \sum_{f \in \mathcal{C}'(W \times X)(\mathbb{1}, \Sigma_Y \mathcal{L}(\pi_2 \circ \pi_1)(x) \multimap \mathcal{L}(\pi_2)(y))} \mathcal{C}'(W \times X)(\mathbb{1}, \mathcal{L}(\pi_2)(x) \multimap \mathcal{L}(f)(y))} \mathcal{L}(W)(w, \Pi_X \mathcal{L}(f)(y)) && \{ \text{comprehension} \} \\
 &\cong \sum_{(f, g) \in \mathcal{C}'(W \times X)(\mathbb{1}, \Sigma_Y \mathcal{L}(\pi_2 \circ \pi_1)(x) \multimap \mathcal{L}(\pi_2)(y))} \mathcal{L}(W)(w, \Pi_X \mathcal{L}(f)(y)) && \{ \text{strong } \Sigma\text{-types in } \mathcal{C}' \} \\
 &= \sum_{(f, g) \in \mathcal{C}'(W \times X)(\mathbb{1}, \Sigma_Y \mathcal{L}(\pi_2 \circ \pi_1)(x) \multimap \mathcal{L}(\pi_2)(y))} \mathcal{L}(W)(w, \Pi_X \mathcal{L}(\text{ev1} \circ ((f, g), \pi_2))((y))) && \{ \text{definition ev1} \} \\
 &= \sum_{(f, g) \in \mathcal{C}'(W \times X)(\mathbb{1}, \Sigma_Y \mathcal{L}(\pi_2 \circ \pi_1)(x) \multimap \mathcal{L}(\pi_2)(y))} \mathcal{L}(W)(w, \Pi_X \mathcal{L}(((f, g), \pi_2))(\mathcal{L}(\text{ev1})(y))) && \{ \text{functoriality of } \mathcal{L} \} \\
 &= \sum_{(f, g) \in \mathcal{C}'(W \times X)(\mathbb{1}, \Sigma_Y \mathcal{L}(\pi_2 \circ \pi_1)(x) \multimap \mathcal{L}(\pi_2)(y))} \mathcal{L}(W)(w, \mathcal{L}((f, g))(\Pi_X \mathcal{L}(\text{ev1})(y))) && \{ \text{Beck-Chevalley for } \Pi\text{-types} \} \\
 &\cong \sum_{h \in \mathcal{C}'(W \times X)(\mathbb{1}, \Sigma_Y \mathcal{L}(\pi_2 \circ \pi_1)(x) \multimap \mathcal{L}(\pi_2)(y))} \mathcal{L}(W)(w, \mathcal{L}(h)(\Pi_X \mathcal{L}(\text{ev1})(y))) && \{ \text{strong } \Sigma\text{-types in } \mathcal{C}' \} \\
 &= \sum_{h \in \mathcal{C}'(W \times X)(\mathcal{L}(\pi_1)(\mathbb{1}), \Sigma_Y \mathcal{L}(\pi_2 \circ \pi_1)(x) \multimap \mathcal{L}(\pi_2)(y))} \mathcal{L}(W)(w, \mathcal{L}(h)(\Pi_X \mathcal{L}(\text{ev1})(y))) && \{ \text{indexed } \mathbb{1} \text{ in } \mathcal{C}' \} \\
 &\cong \sum_{h \in \mathcal{C}'(W)(\mathbb{1}, \Pi_X \Sigma_Y \mathcal{L}(\pi_2 \circ \pi_1)(x) \multimap \mathcal{L}(\pi_2)(y))} \mathcal{L}(W)(w, \mathcal{L}(h)(\Pi_X \mathcal{L}(\text{ev1})(y))) && \{ \Pi\text{-types in } \mathcal{C}' \} \\
 &\cong \sum_{h \in \mathcal{C}(W, \Pi_X \Sigma_Y \mathcal{L}(\pi_1)(x) \multimap \mathcal{L}(\pi_2)(y))} \mathcal{L}(W)(w, \mathcal{L}(h)(\Pi_X \mathcal{L}(\text{ev1})(y))) && \{ \text{comprehension} \} \\
 &= \Sigma_{\mathcal{C}} \mathcal{L}((W, w), (\Pi_X \Sigma_Y \mathcal{L}(\pi_1)(x) \multimap \mathcal{L}(\pi_2)(y), \Pi_X \mathcal{L}(\text{ev1})(y))) \\
 &= \Sigma_{\mathcal{C}} \mathcal{L}((W, w), (X, x) \Rightarrow (Y, y)). \quad \square
 \end{aligned}$$

Codually, we have

Theorem 26. For a Σ -bimodel \mathcal{L} for function types, $\Sigma_{\mathcal{C}} \mathcal{L}^{\text{op}}$ has exponential:

$$(X, x) \Rightarrow (Y, y) = (\Pi_X \Sigma_Y \mathcal{L}(\pi_2)(y) \multimap \mathcal{L}(\pi_1)(x), \Sigma_X \mathcal{L}(\text{ev1})(y)).$$

Note that these exponentials are not fibered over \mathcal{C} in the sense that the projection functors $\Sigma_{\mathcal{C}} \mathcal{L} \rightarrow \mathcal{C}$ and $\Sigma_{\mathcal{C}} \mathcal{L}^{\text{op}} \rightarrow \mathcal{C}$ are generally not cartesian closed functors. This is in contrast with the interpretation of all other type formers we consider in this paper.

6.5 Coproducts in total categories

We, now, study the coproducts in the total categories $\Sigma_{\mathcal{C}} \mathcal{L}$ and $\Sigma_{\mathcal{C}} \mathcal{L}^{\text{op}}$. We are particularly interested in the case of *extensive indexed categories*, a notion introduced in Section 6.6. For future reference, we start by recalling the general case: see, for instance, Gray (1966) for a basic reference on properties of the total categories.

Proposition 27 (Initial object in $\Sigma_{\mathcal{C}} \mathcal{L}$). Let $\mathcal{L} : \mathcal{C}^{\text{op}} \rightarrow \mathbf{Cat}$ be a strictly indexed category. We assume that

- (i) \mathcal{C} has initial object $\mathbb{0}$;
- (ii) $\mathcal{L}(\mathbb{0})$ has initial object, denoted, by abuse of language, by $\mathbb{0}$.

In this case, $(\mathbb{0}, \mathbb{0})$ is the initial object of $\Sigma_{\mathcal{C}} \mathcal{L}$.

Proof. Assuming the hypothesis above, given any object $(Y, y) \in \Sigma_{\mathcal{C}} \mathcal{L}$,

$$\begin{aligned}
 &\Sigma_{\mathcal{C}} \mathcal{L}((\mathbb{0}, \mathbb{0}), (Y, y)) \\
 &= \coprod_{n \in \mathcal{C}(\mathbb{0}, Y)} \mathcal{L}(\mathbb{0})(\mathbb{0}, \mathcal{L}(n)(y)) && \{ \text{by definition} \}
 \end{aligned}$$

$$\begin{aligned} &\cong \coprod_{n \in \mathcal{C}(\mathbb{0}, Y)} \mathbb{1} && \{ \mathbb{0} \text{ initial in } \mathcal{L}(\mathbb{0}) \} \\ &\cong \mathbb{1}. && \{ \mathbb{0} \text{ initial in } \mathcal{C} \} \end{aligned} \quad \square$$

Proposition 28 (Coproducts in $\Sigma_{\mathcal{C}}\mathcal{L}$). *Let $\mathcal{L} : \mathcal{C}^{\text{op}} \rightarrow \mathbf{Cat}$ be a strictly indexed category. We assume that*

- (i) $((W_i, w_i))_{i \in I}$ is family of objects of $\Sigma_{\mathcal{C}}\mathcal{L}$;
- (ii) the category \mathcal{C} has the coproduct:

$$\left(W_t \xrightarrow{\iota_{W_t}} \coprod_{i \in I} W_i \right)_{t \in I} \tag{6}$$

of the objects in $((W_i, w_i))_{i \in I}$;

- (iii) there is an adjunction $\mathcal{L}(\iota_{W_i})! \dashv \mathcal{L}(\iota_{W_i})$ for each $i \in I$;

- (iv) $\mathcal{L}\left(\coprod_{i \in I} W_i\right)$ has the coproduct $\coprod_{i \in I} \mathcal{L}(\iota_{W_i})!(w_i)$ of the objects $(\mathcal{L}(\iota_{W_i})!(w_i))_{i \in I}$.

In this case,

$$\left(\coprod_{i \in I} W_i, \quad \coprod_{i \in I} \mathcal{L}(\iota_{W_i})!(w_i) \right)$$

is the coproduct of the objects $((W_i, w_i))_{i \in I}$ in $\Sigma_{\mathcal{C}}\mathcal{L}$.

Proof. Assuming the hypothesis above, given any object $(Y, y) \in \Sigma_{\mathcal{C}}\mathcal{L}$,

$$\begin{aligned} &\prod_{i \in I} \Sigma_{\mathcal{C}}\mathcal{L}((W_i, w_i), (Y, y)) \\ &= \prod_{i \in I} \left(\coprod_{n \in \mathcal{C}(W_i, Y)} \mathcal{L}(W_i)(w_i, \mathcal{L}(n)(y)) \right) && \{ \text{by definition} \} \\ &\cong \coprod_{(n_i)_{i \in I} \in \prod_{i \in I} \mathcal{C}(W_i, Y)} \left(\prod_{i \in I} \mathcal{L}(W_i)(w_i, \mathcal{L}(n_i)(y)) \right) && \{ \text{distributivity} \} \\ &\cong \coprod_{h \in \mathcal{C}(\coprod_{i \in I} W_i, Y)} \left(\prod_{i \in I} \mathcal{L}(W_i)(w_i, \mathcal{L}(h \circ \iota_{W_i})(y)) \right) && \{ \text{coprod. univ. property} \} \\ &\cong \coprod_{h \in \mathcal{C}(\coprod_{i \in I} W_i, Y)} \left(\prod_{i \in I} \mathcal{L}(W_i)(w_i, \mathcal{L}(\iota_{W_i}) \circ \mathcal{L}(h)(y)) \right) \\ &\cong \coprod_{h \in \mathcal{C}(\coprod_{i \in I} W_i, Y)} \left(\prod_{i \in I} \mathcal{L}\left(\coprod_{i \in I} W_i\right)(\mathcal{L}(\iota_{W_i})!(w_i), \mathcal{L}(h)(y)) \right) && \{ \text{adjunctions} \} \\ &\cong \coprod_{h \in \mathcal{C}(\coprod_{i \in I} W_i, Y)} \left(\mathcal{L}\left(\coprod_{i \in I} W_i\right)\left(\coprod_{i \in I} \mathcal{L}(\iota_{W_i})!(w_i), \mathcal{L}(h)(y)\right) \right) && \{ \text{coprod. univ. property} \} \end{aligned}$$

$$= \Sigma_C \mathcal{L} \left(\left(\coprod_{i \in I} W_i, \prod_{i \in I} \mathcal{L}(t_{W_i})(w_i) \right), (Y, y) \right). \quad \{ \text{coprod. univ. property} \}$$

□

Codually, we get results on the initial objects and coproducts in the category $\Sigma_C \mathcal{L}^{\text{op}}$ below.

Corollary 29 (Initial object in $\Sigma_C \mathcal{L}^{\text{op}}$). *Let $\mathcal{L} : \mathcal{C}^{\text{op}} \rightarrow \mathbf{Cat}$ be a strictly indexed category. We assume that*

- (i) \mathcal{C} has initial object $\mathbb{0}$;
- (ii) $\mathcal{L}(\mathbb{0})$ has terminal object $\mathbb{1}$.

In this case, $(\mathbb{0}, \mathbb{1})$ is the initial object of $\Sigma_C \mathcal{L}$.

Corollary 30 (Coproducts in $\Sigma_C \mathcal{L}^{\text{op}}$). *Let $\mathcal{L} : \mathcal{C}^{\text{op}} \rightarrow \mathbf{Cat}$ be a strictly indexed category. We assume that*

- (i) $((W_i, w_i))_{i \in I}$ is family of objects of $\Sigma_C \mathcal{L}$;
- (ii) the category \mathcal{C} has the coproduct:

$$\left(W_t \xrightarrow{t_{W_t}} \coprod_{i \in I} W_i \right)_{t \in I} \tag{7}$$

of the objects in $((W_i, w_i))_{i \in I}$;

- (iii) there is an adjunction $\mathcal{L}(t_{W_i}) \dashv \mathcal{L}(t_{W_i})^*$ for each $i \in I$;

- (iv) $\mathcal{L} \left(\coprod_{i \in I} W_i \right)$ has the product $\prod_{i \in I} \mathcal{L}(t_{W_i})^*(w_i)$ of the objects $(\mathcal{L}(t_{W_i})^*(w_i))_{i \in I}$.

In this case,

$$\left(\coprod_{i \in I} W_i, \prod_{i \in I} \mathcal{L}(t_{W_i})^*(w_i) \right)$$

is the coproduct of the objects $((W_i, w_i))_{i \in I}$ in $\Sigma_C \mathcal{L}^{\text{op}}$.

6.6 Extensive indexed categories and coproducts in total categories

We introduce a special property that fits our context well. We call this property *extensivity* because it generalizes the concept of *extensive categories* (see Section 6.12 for the notion of extensive category).

As we will show, the property of *extensivity* is a crucial requirement for our models. One significant advantage of this property is that it allows us to easily construct coproducts in the total categories, even under lenient conditions. We demonstrate this in Theorem 35.

- We assume that the category \mathcal{C} has *finite coproducts*. Given $W, X \in \mathcal{C}$, we denote by:

$$W \xrightarrow{t_1=t_W} W \sqcup X \xleftarrow{t_2=t_X} X \tag{8}$$

the coproduct (and coprojections) in \mathcal{C} , and by $\mathbb{0}$ the initial object of \mathcal{C} .

Definition 31 (Extensive indexed categories). *We call an indexed category $\mathcal{L} : \mathcal{C}^{\text{op}} \rightarrow \mathbf{Cat}$ extensive if, for any $(W, X) \in \mathcal{C} \times \mathcal{C}$, the unique functor:*

$$\mathcal{L}(W \sqcup X) \xrightarrow{(\mathcal{L}(i_W), \mathcal{L}(i_X))} \mathcal{L}(W) \times \mathcal{L}(X) \tag{9}$$

induced by the functors:

$$\mathcal{L}(W) \xleftarrow{\mathcal{L}(i_W)} \mathcal{L}(W \sqcup X) \xrightarrow{\mathcal{L}(i_X)} \mathcal{L}(X) \tag{10}$$

is an equivalence. In this case, for each $(W, X) \in \mathcal{C} \times \mathcal{C}$, we denote by:

$$\mathcal{S}^{(W, X)} : \mathcal{L}(W) \times \mathcal{L}(X) \rightarrow \mathcal{L}(W \sqcup X) \tag{11}$$

an inverse equivalence of $(\mathcal{L}(i_W), \mathcal{L}(i_X))$.

Since the products of \mathcal{C}^{op} are the coproducts of \mathcal{C} , the extensive condition described above is equivalent to say that the (pseudo)functor $\mathcal{L} : \mathcal{C}^{\text{op}} \rightarrow \mathbf{Cat}$ preserves binary (bicategorical) products (up to equivalence).

Since our cases of interest are strict, this leads us to consider *strict extensivity*, that is to say, *whenever we talk about extensive strictly indexed categories, we are assuming that (9) is invertible*. In this case, it is even clearer that extensivity coincides with the well-known notion of preservation of binary products.

Lemma 32 (Extensive strictly indexed categories). *Let $\mathcal{L} : \mathcal{C}^{\text{op}} \rightarrow \mathbf{Cat}$ be an indexed category. \mathcal{L} is strictly extensive if, and only if, \mathcal{L} is a functor that preserves binary products.*

Recall that, in general, *preservation of binary products implies preservation of preterminal objects*; see, for instance, Lucatelli Nunes (2022, Remark 4.14). Lemma 33 is the appropriate analog of this observation suitably applied to the context of extensive indexed categories. Moreover, Lemma 33 can be seen as a generalization of Carboni et al. (1993, Proposition 2.8).

Lemma 33 (Preservation of terminal objects). *Let $\mathcal{L} : \mathcal{C}^{\text{op}} \rightarrow \mathbf{Cat}$ be an extensive indexed category which is not (naturally isomorphic to the functor) constantly equal to $\mathbb{0}$. The unique functor*

$$\mathcal{L}(\mathbb{0}) \rightarrow \mathbb{1} \tag{12}$$

is an equivalence. If, furthermore, (9) is an isomorphism, then (12) is invertible.

Proof. Firstly, given any $X \in \mathcal{C}$ such that $\mathcal{L}(X)$ is not (isomorphic to) the initial object of \mathbf{Cat} , we have that $\mathcal{L}(i_X : \mathbb{0} \rightarrow X)$ is a functor from $\mathcal{L}(X)$ to $\mathcal{L}(\mathbb{0})$. Hence, $\mathcal{L}(\mathbb{0})$ is not isomorphic to the initial category as well.

Secondly, since $i_{\mathbb{0}} : \mathbb{0} \rightarrow \mathbb{0} \sqcup \mathbb{0}$ is an isomorphism, $(\mathcal{L}(i_{\mathbb{0}}), \mathcal{L}(i_{\mathbb{0}}))$ is an equivalence and

$$\begin{array}{ccc} \mathcal{L}(\mathbb{0} \sqcup \mathbb{0}) & \xrightarrow{(\mathcal{L}(i_{\mathbb{0}}), \mathcal{L}(i_{\mathbb{0}}))} & \mathcal{L}(\mathbb{0}) \times \mathcal{L}(\mathbb{0}) \xrightarrow{\pi_{\mathcal{L}(\mathbb{0})}} \mathcal{L}(\mathbb{0}), \\ & \searrow & \uparrow \\ & & \mathcal{L}(i_{\mathbb{0}}) \end{array} \tag{13}$$

we conclude that $\pi_{\mathcal{L}(\mathbb{0})}$ is an equivalence. This proves that $\mathcal{L}(\mathbb{0}) \rightarrow \mathbb{1}$ is an equivalence by Appendix A, Lemma 132. □

We proceed to study the cocartesian structure of $\Sigma_{\mathcal{C}}\mathcal{L}$ (and $\Sigma_{\mathcal{C}}\mathcal{L}^{\text{op}}$) when \mathcal{L} is extensive. We start by proving in Theorem 34 that, in the case of extensive indexed categories, the hypothesis of Proposition 27 always holds.

Theorem 34. *Let $\mathcal{L} : \mathcal{C}^{\text{op}} \rightarrow \mathbf{Cat}$ be an extensive (strictly) indexed category. Assume that X is an object of \mathcal{C} such that $\mathcal{L}(X)$ has initial object $\mathbb{0}$. In this case, for any $W \in \mathcal{C}$, we have an adjunction:*

$$\begin{aligned}
 &\cong \left(W \sqcup X, \mathcal{S}^{(W,X)} \circ (\text{id}_{\mathcal{L}(W)}, \mathbb{0}) (w) \sqcup \mathcal{S}^{(W,X)} \circ (\mathbb{0}, \text{id}_{\mathcal{L}(X)}) (x) \right) \\
 &\cong (W \sqcup X, \mathcal{L}(t_W)!(w) \sqcup \mathcal{L}(t_X)!(x)) \quad \{ \text{Theorem 34} \} \\
 &\cong (W, w) \sqcup (X, x). \quad \{ \text{Proposition 28} \}
 \end{aligned}$$

□

In particular, finite coproducts in $\Sigma_{\mathcal{C}}\mathcal{L}$ are fibered in the sense that the projection functor $\Sigma_{\mathcal{C}}\mathcal{L} \rightarrow \mathcal{C}$ preserves them, on the nose.

Codually, we have:

Corollary 36 (Cocartesian structure of $\Sigma_{\mathcal{C}}\mathcal{L}^{\text{op}}$). *Let $\mathcal{L} : \mathcal{C}^{\text{op}} \rightarrow \mathbf{Cat}$ be an extensive strictly indexed category, with terminal objects $\mathbb{1} \in \mathcal{L}(W)$ for each $W \in \mathcal{C}$. In this case, the category $\Sigma_{\mathcal{C}}\mathcal{L}^{\text{op}}$ has (fibred) initial object $\mathbb{0} = (\mathbb{0}, \mathbb{1}) \in \Sigma_{\mathcal{C}}\mathcal{L}^{\text{op}}$, and (fibred) binary coproduct given by:*

$$(W, w) \sqcup (X, x) = \left(W \sqcup X, \mathcal{S}^{(W,X)}(w, x) \right). \tag{17}$$

Definition 37 (Σ -bimodel for sum types). *A strictly indexed category $\mathcal{L} : \mathcal{C}^{\text{op}} \rightarrow \mathbf{Cat}$ is a Σ -bimodel for sum types if \mathcal{L} is an extensive strictly indexed category such that $\mathcal{L}(W)$ has initial and terminal objects.*

6.7 Distributive property of the total category

We refer the reader to Carboni et al. (1993) and Lack (2012) for the basics on distributive categories.

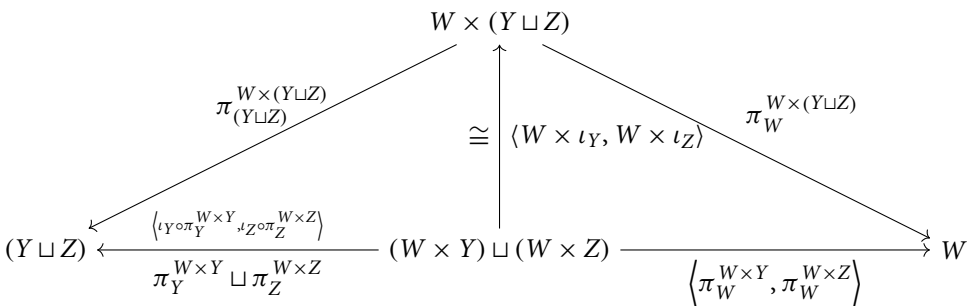
As we proved, $\Sigma_{\mathcal{C}}\mathcal{L}$ is bicartesian closed provided that $\mathcal{L} : \mathcal{C}^{\text{op}} \rightarrow \mathbf{Cat}$ is Σ -bimodel for function types and sum types. Therefore, in this setting, we get that $\Sigma_{\mathcal{C}}\mathcal{L}$ is distributive.

However, even without the assumptions concerning closed structures, whenever we have a Σ -bimodel for sum types, we can inherit distributivity from \mathcal{C} . Namely, we have Theorem 39.

Recall that a category \mathcal{C} with finite products and coproducts is a *distributive category* if, for each triple (W, Y, Z) of objects in \mathcal{C} , the canonical morphism:

$$\left\langle W \times \iota_Y^{Y \sqcup Z}, W \times \iota_Z^{Y \sqcup Z} \right\rangle : (W \times Y) \sqcup (W \times Z) \rightarrow W \times (Y \sqcup Z), \tag{18}$$

induced by $W \times \iota_Y$ and $W \times \iota_Z$ is invertible. It should be noted that, in a such a distributive category \mathcal{C} , for any such a triple (W, Y, Z) of objects in \mathcal{C} , the diagram



commutes. Therefore, we have:

Lemma 38. *Let $\mathcal{L} : \mathcal{C}^{\text{op}} \rightarrow \mathbf{Cat}$ be an extensive strictly indexed category, in which \mathcal{C} is a distributive category. For each triple (W, Y, Z) of objects in \mathcal{C} , the diagrams*

$$\begin{array}{ccc}
 \mathcal{L}(W \times (Y \sqcup Z)) & \xleftarrow{\mathcal{L}(\pi_W^{W \times (Y \sqcup Z)})} & \mathcal{L}(W) \\
 \downarrow \cong & \swarrow \mathcal{L}(\langle \pi_W^{W \times Y}, \pi_W^{W \times Z} \rangle) & \downarrow \\
 \mathcal{L}((W \times Y) \sqcup (W \times Z)) & \xleftarrow{\mathcal{L}(\langle \pi_W^{W \times Y}, \pi_W^{W \times Z} \rangle)} & \mathcal{L}(W) \\
 \uparrow \mathcal{S}^{(W \times Y, W \times Z)} & \searrow (\mathcal{L}(\iota_{W \times Y}), \mathcal{L}(\iota_{W \times Z})) & \downarrow \\
 \mathcal{L}(W \times Y) \times \mathcal{L}(W \times Z) & \xleftarrow{(\mathcal{L}(\pi_W^{W \times Y}), \mathcal{L}(\pi_W^{W \times Z}))} & \mathcal{L}(W)
 \end{array} \tag{19}$$

$$\begin{array}{ccc}
 \mathcal{L}(W \times (Y \sqcup Z)) & \xleftarrow{\mathcal{L}(\pi_{(Y \sqcup Z)}^{W \times (Y \sqcup Z)})} & \mathcal{L}(Y \sqcup Z) \\
 \downarrow \cong & \swarrow \mathcal{L}(\langle \pi_Y^{W \times Y}, \pi_Z^{W \times Z} \rangle) & \downarrow \\
 \mathcal{L}((W \times Y) \sqcup (W \times Z)) & \xleftarrow{\mathcal{L}(\langle \pi_Y^{W \times Y}, \pi_Z^{W \times Z} \rangle)} & \mathcal{L}(Y \sqcup Z) \\
 \uparrow \mathcal{S}^{(W \times Y, W \times Z)} & \searrow (\mathcal{L}(\iota_{W \times Y}), \mathcal{L}(\iota_{W \times Z})) & \downarrow \mathcal{S}^{(Y, Z)} \\
 \mathcal{L}(W \times Y) \times \mathcal{L}(W \times Z) & \xleftarrow{\mathcal{L}(\pi_Y^{W \times Y}) \times \mathcal{L}(\pi_Z^{W \times Z})} & \mathcal{L}(Y) \times \mathcal{L}(Z)
 \end{array} \tag{20}$$

commute.

Theorem 39. Let $\mathcal{L} : \mathcal{C}^{\text{op}} \rightarrow \mathbf{Cat}$ be Σ -bimodel for sum and tuple types, in which \mathcal{C} is a distributive category. In this setting, the category $\Sigma_{\mathcal{C}}\mathcal{L}$ is a distributive category.

Proof. By Proposition 17 and Corollary 35, we have that $\Sigma_{\mathcal{C}}\mathcal{L}$ indeed has finite coproducts and finite products.

Let \mathcal{D} be a category with finite coproducts and products. A category is distributive if the canonical morphisms (18) are invertible. However, by Lack (2012, Theorem 4), the existence of any natural isomorphism $(W \times Y) \sqcup (W \times Z) \cong W \times (Y \sqcup Z)$ implies that \mathcal{D} distributive. Hence, we proceed to prove below that such a natural isomorphism exists in the case of $\Sigma_{\mathcal{C}}\mathcal{L}$, leaving the question of canonicity omitted.

We indeed have the natural isomorphisms in $((W, w), (Y, y), (Z, z)) \in \Sigma_{\mathcal{C}}\mathcal{L} \times \Sigma_{\mathcal{C}}\mathcal{L} \times \Sigma_{\mathcal{C}}\mathcal{L}$:

$$\begin{aligned}
 & (W, w) \times ((Y, y) \sqcup (Z, z)) \\
 & \cong (W, w) \times (Y \sqcup Z, \mathcal{S}^{(Y, Z)}(y, z)) && \{ \text{Corollary 35} \} \\
 & \cong (W \times (Y \sqcup Z), \mathcal{L}(\pi_W)(w) \times \mathcal{L}(\pi_{Y \sqcup Z})\mathcal{S}^{(Y, Z)}(y, z)), && \{ \text{Proposition 17} \}
 \end{aligned}$$

which, by the distributive property of \mathcal{C} , is (naturally) isomorphic to

$$((W \times Y) \sqcup (W \times Z), \mathcal{L}(\langle \iota_Y, \iota_Z \rangle) (\mathcal{L}(\pi_W)(w) \times \mathcal{L}(\pi_{Y \sqcup Z})\mathcal{S}^{(Y, Z)}(y, z))). \tag{21}$$

Moreover, we have the natural isomorphisms

$$\begin{aligned} & \mathcal{L}((W \times_{\iota_Y}, W \times_{\iota_Z}))(\mathcal{L}(\pi_W)(w) \times \mathcal{L}(\pi_{Y \sqcup Z})\mathcal{S}^{(Y,Z)}(y,z)) \\ & \cong \mathcal{L}((W \times_{\iota_Y}, W \times_{\iota_Z}))(\mathcal{L}(\pi_W)(w)) \times \mathcal{L}((W \times_{\iota_Y}, W \times_{\iota_Z}))(\mathcal{L}(\pi_{Y \sqcup Z})\mathcal{S}^{(Y,Z)}(y,z)) && \{ \mathcal{L}((W \times_{\iota_Y}, W \times_{\iota_Z})) \text{ invertible} \} \\ & = \mathcal{S}^{(W \times Y, W \times Z)}(\mathcal{L}(\pi_W)(w), \mathcal{L}(\pi_W)(w)) \times \mathcal{L}((W \times_{\iota_Y}, W \times_{\iota_Z}) \circ \mathcal{L}(\pi_{Y \sqcup Z}) \circ \mathcal{S}^{(Y,Z)}(y,z)) && \{ \text{Diagram (19)} \} i \\ & = \mathcal{S}^{(W \times Y, W \times Z)}(\mathcal{L}(\pi_W)(w), \mathcal{L}(\pi_W)(w)) \times \mathcal{S}^{(W \times Y, W \times Z)}(\mathcal{L}(\pi_Y)(y), \mathcal{L}(\pi_Z)(z)), && \{ \text{Diagram (20)} \} \end{aligned}$$

which is naturally isomorphic to

$$\mathcal{S}^{(W \times Y, W \times Z)}(\mathcal{L}(\pi_W)(w) \times \mathcal{L}(\pi_Y)(y), \mathcal{L}(\pi_W)(w) \times \mathcal{L}(\pi_Z)(z)). \tag{22}$$

since $\mathcal{S}^{(W \times Y, W \times Z)}$ is invertible. Therefore, we have the natural isomorphisms:

$$\begin{aligned} & (W, w) \times ((Y, y) \sqcup (Z, z)) \\ & \cong ((W \times Y) \sqcup (W \times Z), \mathcal{L}((W \times_{\iota_Y}, W \times_{\iota_Z}))(\mathcal{L}(\pi_W)(w) \times \mathcal{L}(\pi_{Y \sqcup Z})\mathcal{S}^{(Y,Z)}(y,z))) && \{ \text{Eq. (21)} \} \\ & \cong ((W \times Y) \sqcup (W \times Z), \mathcal{S}^{(W \times Y, W \times Z)}(\mathcal{L}(\pi_W)(w) \times \mathcal{L}(\pi_Y)(y), \mathcal{L}(\pi_W)(w) \times \mathcal{L}(\pi_Z)(z))) && \{ \text{Eq. (22)} \} \\ & \cong (W \times Y, \mathcal{L}(\pi_W)(w) \times \mathcal{L}(\pi_Y)(y)) \sqcup (W \times Z, \mathcal{L}(\pi_W)(w) \times \mathcal{L}(\pi_Z)(z)) && \{ \text{Corollary 35} \} \\ & ((W, w) \times (Y, y)) \sqcup ((W, w) \times (Z, z)), && \{ \text{Proposition 17} \} \end{aligned}$$

which completes our proof. □

Codually, we have:

Theorem 40. *Let $\mathcal{L} : \mathcal{C}^{\text{op}} \rightarrow \mathbf{Cat}$ be a Σ -bimodel for sum and tuple types, in which \mathcal{C} is a distributive category. Then, we conclude that $\Sigma_{\mathcal{C}}\mathcal{L}^{\text{op}}$ is a distributive category.*

6.8 Extensive property of the total category

As per the definition provided in Carboni et al. (1993, Definition 2.1), a category \mathcal{C} is considered extensive if the basic (codomain) indexed category $\mathcal{C}/- : \mathcal{C}^{\text{op}} \rightarrow \mathbf{Cat}$ is an extensive indexed category as introduced at Definition 31. Recall that every extensive category is distributive (Carboni et al. 1993, Proposition 4.5).

The result below also holds for the nonstrict scenario.

Theorem 41. *Let $\mathcal{L} : \mathcal{C}^{\text{op}} \rightarrow \mathbf{Cat}$ be an extensive strictly indexed category, in which \mathcal{C} is an extensive category. Assume that we have initial objects $0 \in \mathcal{L}(W)$. In this case, the category $\Sigma_{\mathcal{C}}\mathcal{L}$ is extensive and, hence, distributive.*

Proof. We denote by $\mathcal{S}_{\mathcal{L}}^{(W,X)} : \mathcal{L}(W) \times \mathcal{L}(X) \rightarrow \mathcal{L}(W \sqcup X)$ the isomorphisms of the extensive strictly indexed category \mathcal{L} .

The first step is to see that, indeed, $\Sigma_{\mathcal{C}}\mathcal{L}$ has coproducts by Corollary 35. We, then, note that, for each pair (W, w) and (X, x) of objects in $\Sigma_{\mathcal{C}}\mathcal{L}$, we note that, in fact, we have that

$$\mathcal{S}_{\Sigma_{\mathcal{C}}\mathcal{L}/-}^{((W,w),(X,x))} : \Sigma_{\mathcal{C}}\mathcal{L}/(W, w) \times \Sigma_{\mathcal{C}}\mathcal{L}/(X, x) \rightarrow \Sigma_{\mathcal{C}}\mathcal{L}/((W, w) \sqcup (X, x)) \tag{23}$$

defined by the coproduct of the morphisms is an equivalence. Explicitly, given objects $A = ((W_0, w_0), (f : W_0 \rightarrow W, f' : w_0 \rightarrow \mathcal{L}(f) w))$ of $\Sigma_{\mathcal{C}}\mathcal{L}/(W, w)$ and $B = ((X_0, x_0), (g : X_0 \rightarrow X, g' : x_0 \rightarrow \mathcal{L}(g) x))$ of $\Sigma_{\mathcal{C}}\mathcal{L}/(X, x)$, $\mathcal{S}_{\Sigma_{\mathcal{C}}\mathcal{L}/-}^{((W,w),(X,x))}(A, B)$ is given by:

$$\left((W_0 \sqcup X_0, \mathcal{S}_{\mathcal{L}}^{(W,X)}(w_0, x_0)), (f \sqcup g : W_0 \sqcup X_0 \rightarrow W \sqcup X, \mathcal{S}_{\mathcal{L}}^{(W,X)}(f', g')) \right)$$

which is clearly an equivalence given that the functor $((W_0, f), (X_0, g)) \mapsto (W_0 \sqcup X_0, f \sqcup g)$ is an equivalence $\mathcal{C}/W \times \mathcal{C}/X \rightarrow \mathcal{C}/W \sqcup X$. □

Theorem 42. *Let $\mathcal{L} : \mathcal{C}^{\text{op}} \rightarrow \mathbf{Cat}$ be an extensive strictly indexed category, in which \mathcal{C} is an extensive category. Assume that we have terminal objects $\mathbb{1} \in \mathcal{L}(W)$. In this case, the category $\Sigma_{\mathcal{C}}\mathcal{L}^{\text{op}}$ is extensive and, hence, distributive.*

It is worth mentioning that free cocompletions under (finite) coproducts are extensive, as shown in Carboni et al. (1993, Proposition 2.4) for the infinite case. This implies that freely generated models on languages featuring variant types are extensive. Therefore, having an extensive base category \mathcal{C} is a common occurrence in our setting.

6.9 Strictly indexed categories and split fibrations

Before we specialize to our setting of $\mu\nu$ -polynomials, we need to establish and prove general results on parameterized initial algebras (and terminal coalgebras) in the total category of a split fibration (see Sections 6.10 and 6.11).

In order to talk about these results, we need to talk about *strictly indexed functors* and *split fibration functors* and the one-to-one correspondence between them. For this purpose, we shortly recall the equivalence between strict indexed categories and split fibrations below.

Definition 43 (Strictly indexed functor). *Let $\mathcal{L}' : \mathcal{D}^{\text{op}} \rightarrow \mathbf{Cat}$ and $\mathcal{L} : \mathcal{C}^{\text{op}} \rightarrow \mathbf{Cat}$ be two strictly indexed categories. A strictly indexed functor between \mathcal{L}' and \mathcal{L} consists of a pair (\overline{H}, h) in which $\overline{H} : \mathcal{D} \rightarrow \mathcal{C}$ is a functor and*

$$h : \mathcal{L}' \longrightarrow (\mathcal{L} \circ \overline{H}^{\text{op}}) \tag{24}$$

is a natural transformation, where \overline{H}^{op} denotes the image of \overline{H} by op . Given two strictly indexed functors $(\overline{E}, e) : \mathcal{L}'' \rightarrow \mathcal{L}'$ and $(\overline{H}, h) : \mathcal{L}' \rightarrow \mathcal{L}$, the composition is given by:

$$(\overline{HE}, (h_{\overline{E}^{\text{op}}}) \cdot e : \mathcal{L}'' \longrightarrow (\mathcal{L} \circ (\overline{HE})^{\text{op}})). \tag{25}$$

Strictly indexed categories and strictly indexed functors do form a category, denoted herein by $\mathfrak{In}\mathfrak{d}$.

It is well known that the Grothendieck construction provides an equivalence between indexed categories and fibrations. Restricting this to our setting, we get the equivalence:

$$\int : \begin{array}{ccc} \mathfrak{In}\mathfrak{d} & \rightarrow & \text{Sp}\mathfrak{Fib} \\ \mathcal{L} : \mathcal{C}^{\text{op}} \rightarrow \mathbf{Cat} & \mapsto & (\text{P}_{\mathcal{L}} : \Sigma_{\mathcal{C}}\mathcal{L} \rightarrow \mathcal{C}) \\ (\overline{E}, e) & \mapsto & (E, \overline{E}) \end{array}$$

between the category of strictly indexed categories (with strictly indexed functors) and the category of (Grothendieck) split fibrations.

Although not necessary to your work, we refer to Gray (1966) and Johnstone (2002, Theorem 1.3.6) for further details. We explicitly state the relevant part of this result below.

Proposition 44. *Given two strictly indexed categories, $\mathcal{L}' : \mathcal{D}^{\text{op}} \rightarrow \mathbf{Cat}$ and $\mathcal{L} : \mathcal{C}^{\text{op}} \rightarrow \mathbf{Cat}$, there is a bijection between strictly indexed functors:*

$$(\overline{H} : \mathcal{D} \rightarrow \mathcal{C}, h : \mathcal{L}' \longrightarrow (\mathcal{L} \circ \overline{H}^{\text{op}})) : \mathcal{L}' \rightarrow \mathcal{L}$$

and pairs (H, \overline{H}) in which $H : \Sigma_{\mathcal{D}}\mathcal{L}' \rightarrow \Sigma_{\mathcal{C}}\mathcal{L}$ is a functor satisfying the following two conditions.

ΣA) The diagram

$$\begin{array}{ccc}
 \Sigma_{\mathcal{D}}\mathcal{L}' & \xrightarrow{H} & \Sigma_{\mathcal{C}}\mathcal{L} \\
 P_{\mathcal{L}'} \downarrow & & \downarrow P_{\mathcal{L}} \\
 \mathcal{D} & \xrightarrow{\bar{H}} & \mathcal{C}
 \end{array} \tag{26}$$

commutes.

ΣB) For any morphism $(f : X \rightarrow Y, \text{id} : \mathcal{L}'(f)(y) \rightarrow \mathcal{L}'(f)(y))$ between $(X, \mathcal{L}'(f)(y))$ and (Y, y) in $\Sigma_{\mathcal{D}}\mathcal{L}'$,

$$H(f, \text{id}) = (\bar{H}(f), \text{id}) : H(X, \mathcal{L}'(f)(y)) \rightarrow H(Y, y). \tag{27}$$

Proof. Although, as mentioned above, this result is just a consequence of the well-known result about the equivalence between indexed categories and fibrations, we recall below how to construct the bijection.

For each strictly indexed functor $(\bar{H}, h) : \mathcal{L}' \rightarrow \mathcal{L}$, we define

$$H(f : X \rightarrow Y, f' : x \rightarrow \mathcal{L}'(f)y) := (\bar{H}(f), h_X(f')). \tag{28}$$

Reciprocally, given a pair (H, \bar{H}) satisfying (26) and (27), we define

$$h_X(f' : w \rightarrow x) := H((\text{id}_X, f') : (X, w) \rightarrow (X, x)) \tag{29}$$

for each object $X \in \mathcal{D}$ and each morphism $f' : w \rightarrow x$ of $\mathcal{L}'(X)$. □

Definition 45 (Split fibration functor). A pair $(H, \bar{H}) : P_{\mathcal{L}'} \rightarrow P_{\mathcal{L}}$ satisfying (26) and (27) is herein called a split fibration functor. Whenever it is clear from the context, we omit the split fibrations $P_{\mathcal{L}'}$, $P_{\mathcal{L}}$, and the functor \bar{H} .

Following the above, given a strictly indexed functor $(\bar{H}, h) : \mathcal{L}' \rightarrow \mathcal{L}$, we denote

$$\int \mathcal{L} = (P_{\mathcal{L}} : \Sigma_{\mathcal{C}}\mathcal{L} \rightarrow \mathcal{C}) \\
 \int (\bar{H}, h) = (H, \bar{H})$$

in which $H(f : X \rightarrow Y, f' : x \rightarrow \mathcal{L}'(f)(y)) = (\bar{H}(f), h_X(f'))$.

Let $\mathcal{L}' : \mathcal{D}^{\text{op}} \rightarrow \mathbf{Cat}$ and $\mathcal{L} : \mathcal{C}^{\text{op}} \rightarrow \mathbf{Cat}$ be strictly indexed categories. We denote by $\mathcal{L}' \underline{\times} \mathcal{L}$ the product of the strict indexed categories in $\mathfrak{In}\mathfrak{d}$. Explicitly,

$$\begin{aligned}
 \mathcal{L}' \underline{\times} \mathcal{L} : (\mathcal{D} \times \mathcal{C})^{\text{op}} &\rightarrow \mathbf{Cat} \\
 (X, Y) &\mapsto \mathcal{L}'(X) \times \mathcal{L}(Y) \\
 (f, g) &\mapsto \mathcal{L}'(f) \times \mathcal{L}(g).
 \end{aligned}$$

It should be noted that

$$\left(\int \mathcal{L}' \underline{\times} \mathcal{L}\right) \cong \left(\int \mathcal{L}'\right) \times \left(\int \mathcal{L}\right) = (P_{\mathcal{L}'} \times P_{\mathcal{L}} : (\Sigma_{\mathcal{D}}\mathcal{L}') \times (\Sigma_{\mathcal{C}}\mathcal{L}) \rightarrow \mathcal{D} \times \mathcal{C}), \tag{30}$$

which means that the product in $\mathfrak{Sp}\mathfrak{fib}$ coincides with the usual product of functors $P_{\mathcal{L}} \times P_{\mathcal{L}'}$. Moreover, given indexed functors $(\bar{H}, h) : \mathcal{H} \rightarrow \mathcal{H}'$ and $(\bar{E}, e) : \mathcal{L} \rightarrow \mathcal{L}'$, we have that

$$(\bar{H}, h) \underline{\times} (\bar{E}, e) = (\bar{H} \times \bar{E}, h \times e)$$

and, since the product of split fibrations is given by the usual product of functors:

$$\int ((\bar{H}, h) \underline{\times} (\bar{E}, e)) = \left(\int (\bar{H}, h)\right) \times \left(\int (\bar{E}, e)\right). \tag{31}$$

Codually, given a strictly indexed category $\mathcal{L} : \mathcal{C}^{\text{op}} \rightarrow \mathbf{Cat}$, we have the Grothendieck codual construction:

$$f^{\text{op}} \mathcal{L} = (\mathbf{P}_{\mathcal{L}^{\text{op}}} : \Sigma_{\mathcal{C}} \mathcal{L}^{\text{op}} \rightarrow \mathcal{C}), \quad f^{\text{op}} (\overline{H}, h) = (H, \overline{H})$$

in which $H(f : X \rightarrow Y, f' : \mathcal{L}(f)(y) \rightarrow x) = (\overline{H}(f), h_X(f'))$. This construction gives an equivalence between the indexed categories and split op-fibrations (if we consider the opposite of $f^{\text{op}} \mathcal{L}$). We, of course, have the codual observations above.

6.10 General result on initial algebras in total categories

In order to study the $\mu\nu$ -polynomials of total categories in our setting in Section 6.12, we start by establishing general results about parameterized initial algebras in the Grothendieck construction of split fibrations. More precisely, in Theorem 47, we investigate when a total category $\Sigma_{\mathcal{C}} \mathcal{L}$ has the parameterized initial algebra of a split fibration functor:

$$H : (\Sigma_{\mathcal{D}} \mathcal{L}') \times (\Sigma_{\mathcal{C}} \mathcal{L}) \rightarrow \Sigma_{\mathcal{C}} \mathcal{L}. \tag{32}$$

We start by studying initial algebras of strictly indexed endofunctors:

Theorem 46 (Initial algebras of strictly indexed endofunctors). *Let (\overline{E}, e) be a strictly indexed endofunctor on $\mathcal{L} : \mathcal{C}^{\text{op}} \rightarrow \mathbf{Cat}$ and $E : \Sigma_{\mathcal{C}} \mathcal{L} \rightarrow \Sigma_{\mathcal{C}} \mathcal{L}$ the corresponding split fibration endofunctor. Assume that*

- (e1) *the initial \overline{E} -algebra $(\mu \overline{E}, \text{in}_{\overline{E}})$ exists;*
- (e2) *the initial $(\mathcal{L}(\text{in}_{\overline{E}})^{-1} e_{\mu \overline{E}})$ -algebra $(\mu (\mathcal{L}(\text{in}_{\overline{E}})^{-1} e_{\mu \overline{E}}), \text{in}_{(\mathcal{L}(\text{in}_{\overline{E}})^{-1} e_{\mu \overline{E}})})$ exists.*

Denoting by \underline{e} the endofunctor $\mathcal{L}(\text{in}_{\overline{E}})^{-1} e_{\mu \overline{E}}$ on $\mathcal{L}(\mu \overline{E})$, the initial E -algebra exists and is given by:

$$\mu E = (\mu \overline{E}, \mu \underline{e}), \quad \text{in}_E = (\text{in}_{\overline{E}}, \mathcal{L}(\text{in}_{\overline{E}})(\text{in}_{\underline{e}})). \tag{33}$$

Moreover, for each E -algebra:

$$((Y, y), (\xi, \xi') : E(Y, y) \rightarrow (Y, y)) = ((Y, y), (\xi : \overline{E}(Y) \rightarrow Y, \xi' : e_Y(y) \rightarrow \mathcal{L}(\xi)(y))),$$

we have that

$$\text{fold}_E(\xi, \xi') = \left(\text{fold}_{\overline{E}} \xi, \text{fold}_{\underline{e}} \left(\mathcal{L} \left(\overline{E}(\text{fold}_{\overline{E}} \xi) \cdot \text{in}_{\overline{E}}^{-1} \right) (\xi') \right) \right). \tag{34}$$

Proof. In fact, under the hypothesis above, given an E -algebra:

$$(\xi : \overline{E}(Y) \rightarrow Y, \xi' : e_Y(y) \rightarrow \mathcal{L}(\xi)(y))$$

on (Y, y) , we have that there is a unique morphism:

$$\left(\text{fold}_{\underline{e}} \mathcal{L} \left(\overline{E}(\text{fold}_{\overline{E}} \xi) \cdot \text{in}_{\overline{E}}^{-1} \right) (\xi') \right) : \mu \underline{e} \rightarrow \mathcal{L}(\text{fold}_{\overline{E}} \xi)(y)$$

in $\mathcal{L}(\mu \overline{E})$ such that

$$\begin{array}{ccc}
 \underline{e}(\underline{\mu e}) & \xrightarrow{\underline{e}(\text{fold}_{\underline{e}} \mathcal{L}(\overline{E}(\text{fold}_{\overline{E}}\xi) \cdot \text{in}_{\overline{E}}^{-1})(\xi'))} & \underline{e} \circ \mathcal{L}(\text{fold}_{\overline{E}}\xi)(y) \\
 \downarrow \text{in}_{\underline{e}} & & \parallel \\
 & & \mathcal{L}(\overline{E}(\text{fold}_{\overline{E}}\xi) \cdot \text{in}_{\overline{E}}^{-1}) \circ e_Y(y) \\
 & & \downarrow \mathcal{L}(\overline{E}(\text{fold}_{\overline{E}}\xi) \cdot \text{in}_{\overline{E}}^{-1})(\xi') \\
 & & \mathcal{L}(\xi \cdot \overline{E}(\text{fold}_{\overline{E}}\xi) \cdot \text{in}_{\overline{E}}^{-1})(y) \\
 & & \parallel \\
 \underline{\mu e} & \xrightarrow{(\text{fold}_{\underline{e}} \mathcal{L}(\overline{E}(\text{fold}_{\overline{E}}\xi) \cdot \text{in}_{\overline{E}}^{-1})(\xi'))} & \mathcal{L}(\text{fold}_{\overline{E}}\xi)(y)
 \end{array}$$

commutes. Since $\mathcal{L}(\text{in}_{\overline{E}})$ is invertible, this implies that

$$(\text{fold}_{\underline{e}} \mathcal{L}(\overline{E}(\text{fold}_{\overline{E}}\xi) \cdot \text{in}_{\overline{E}}^{-1})(\xi')) : \underline{\mu e} \rightarrow \mathcal{L}(\text{fold}_{\overline{E}}\xi)(y)$$

is the unique morphism in $\mathcal{L}(\overline{E}(\underline{\mu E}))$ such that

$$\begin{array}{ccc}
 e_{\underline{\mu E}}(\underline{\mu e}) & \xrightarrow{e_{\underline{\mu E}}(\text{fold}_{\underline{e}} \mathcal{L}(\overline{E}(\text{fold}_{\overline{E}}\xi) \cdot \text{in}_{\overline{E}}^{-1})(\xi'))} & e_{\underline{\mu E}} \circ \mathcal{L}(\text{fold}_{\overline{E}}\xi)(y) \\
 \downarrow \mathcal{L}(\text{in}_{\overline{E}})(\text{in}_{\underline{e}}) & & \parallel \\
 & & \mathcal{L}(\overline{E}(\text{fold}_{\overline{E}}\xi)) \circ e_Y(y) \\
 & & \downarrow \mathcal{L}(\overline{E}(\text{fold}_{\overline{E}}\xi))(\xi') \\
 & & \mathcal{L}(\overline{E}(\text{fold}_{\overline{E}}\xi)) \circ \mathcal{L}(\xi)(y) \\
 & & \parallel \\
 \mathcal{L}(\text{in}_{\overline{E}})(\underline{\mu e}) & \xrightarrow{\mathcal{L}(\text{in}_{\overline{E}})(\text{fold}_{\underline{e}} \mathcal{L}(\overline{E}(\text{fold}_{\overline{E}}\xi) \cdot \text{in}_{\overline{E}}^{-1})(\xi'))} & \mathcal{L}(\text{in}_{\overline{E}}) \circ \mathcal{L}(\text{fold}_{\overline{E}}\xi)(y)
 \end{array}$$

commutes. Finally, by the above and the universal property of $\text{fold}_{\overline{E}}\xi$, this completes the proof that

$$\mathbf{u} = \left(\text{fold}_{\overline{E}}\xi, \left(\text{fold}_{\underline{e}} \mathcal{L}(\overline{E}(\text{fold}_{\overline{E}}\xi) \cdot \text{in}_{\overline{E}}^{-1})(\xi') \right) \right) \tag{35}$$

is the unique morphism in $\Sigma_C \mathcal{L}$ such that

$$(\xi, \xi') \circ E(\mathbf{u}) = \mathbf{u} \circ (\text{in}_{\overline{E}}, \mathcal{L}(\text{in}_{\overline{E}})(\text{in}_{\underline{e}})).$$

This completes the proof that $((\underline{\mu E}, \underline{\mu e}), (\text{in}_{\overline{E}}, \mathcal{L}(\text{in}_{\overline{E}})(\text{in}_{\underline{e}})))$ is the initial object of $E\text{-Alg}$, and that $\text{fold}_{E((Y, y), (\xi, \xi'))} = \mathbf{u}$. □

Let $\mathcal{L} : \mathcal{C}^{\text{op}} \rightarrow \mathbf{Cat}$, $\mathcal{L}' : \mathcal{D}^{\text{op}} \rightarrow \mathbf{Cat}$ be strictly indexed categories as above. We denote by $\mathcal{L}' \times \mathcal{L} : (\mathcal{D} \times \mathcal{C})^{\text{op}} \rightarrow \mathbf{Cat}$ the product of the indexed categories (see Section 6.9). An object of $\Sigma_{\mathcal{D} \times \mathcal{C}}(\mathcal{L}' \times \mathcal{L}) \cong (\Sigma_{\mathcal{D}}\mathcal{L}') \times (\Sigma_{\mathcal{C}}\mathcal{L})$ can be seen as a quadruple $((X, x), (W, w))$ in which $x \in \mathcal{L}'(X)$ and $w \in \mathcal{L}(W)$. Moreover, a morphism between objects $((X_0, x_0), (W_0, w_0))$ and $((X_1, x_1), (W_1, w_1))$ consists of a quadruple $((f, f'), (g, g'))$ in which $(f, g) : (X_0, W_0) \rightarrow (X_1, W_1)$ is a morphism in $\mathcal{D} \times \mathcal{C}$, and $(f', g') : (x_0, w_0) \rightarrow (\mathcal{L}'(f)(x_1), \mathcal{L}(g)(w_1))$ is a morphism in $\mathcal{L}'(X_0) \times \mathcal{L}(W_0)$.

Given a strictly indexed functor $(\overline{H}, h) : \mathcal{L}' \times \mathcal{L} \rightarrow \mathcal{L}$ and an object (X, x) of $(\Sigma_{\mathcal{D}}\mathcal{L}')$, we can consider the restriction $(\overline{H}^X, h^{(X,x)})$ in which $\overline{H}^X = \overline{H}(X, -)$ and $h^{(X,x)} : \mathcal{L} \rightarrow (\mathcal{L} \circ \overline{H}^X)$ is pointwise defined by:

$$h_Y^{(X,x)} : \mathcal{L}(Y) \rightarrow \mathcal{L} \circ \overline{H}^X(Y)$$

$$f' : y \rightarrow z \mapsto h_{(X,Y)}(x, f')$$

in which we denote by $(X, Y) \in \mathcal{D} \times \mathcal{C}$. To be consistent with the notation previously introduced (in Proposition 4), we also denote by $h_{(X,Y)}^x$ the morphism $h_Y^{(X,x)}$ above.

As a consequence of Theorem 46, we have that, under suitable conditions, parameterized initial algebras of split fibration functors are split fibration functors, namely, we have:

Theorem 47 (Parameterized initial algebras are split fibration functors). *Let (\overline{H}, h) be a strictly indexed functor from $\mathcal{L}' \times \mathcal{L} : (\mathcal{D} \times \mathcal{C})^{\text{op}} \rightarrow \mathbf{Cat}$ to $\mathcal{L} : \mathcal{C}^{\text{op}} \rightarrow \mathbf{Cat}$, and*

$$H : (\Sigma_{\mathcal{D}}\mathcal{L}') \times (\Sigma_{\mathcal{C}}\mathcal{L}) \rightarrow \Sigma_{\mathcal{C}}\mathcal{L}$$

the corresponding split fibration functor. Assume that:

- (h1) *for each object X of \mathcal{D} , the initial \overline{H}^X -algebra $(\mu\overline{H}^X, \text{in}_{\overline{H}^X})$ exists;*
- (h2) *for each object (X, x) in $\Sigma_{\mathcal{D}}\mathcal{L}'$, denoting by h_X the functor:*

$$\mathcal{L}(\text{in}_{\overline{H}^X})^{-1} h_{(X, \mu\overline{H}^X)} : \mathcal{L}'(X) \times \mathcal{L}(\mu\overline{H}^X) \rightarrow \mathcal{L}(\mu\overline{H}^X) \tag{36}$$

is such that the initial h_X^x -algebra $(\mu h_X^x, \text{in}_{h_X^x})$ exists;

- (h3) *for each morphism $g : X \rightarrow Y$ in \mathcal{D} and $y \in \mathcal{L}'(Y)$, Eq. (37) holds*

$$\mathcal{L}(\mu\overline{H}(g))(\text{in}_{h_Y^y}) = \text{in}_{h_X^{\mathcal{L}'(g)(y)}} \tag{37}$$

In this setting, the parameterized initial algebra $\mu H : \Sigma_{\mathcal{D}}\mathcal{L}' \rightarrow \Sigma_{\mathcal{C}}\mathcal{L}$ exists and is a split fibration functor.

Proof. Assuming the hypothesis, we conclude that, for each (X, x) in $\Sigma_{\mathcal{D}}\mathcal{L}'$, the category $\Sigma_{\mathcal{C}}\mathcal{L}$ has the initial $H^{(X,x)}$ -algebra, by Theorem 46. Hence, we have that

$$\mu H : \Sigma_{\mathcal{D}}\mathcal{L}' \rightarrow \Sigma_{\mathcal{C}}\mathcal{L}$$

exists by Proposition 4. More precisely, given a morphism $(f, f') : (X, x) \rightarrow (Y, y)$ in $\Sigma_{\mathcal{D}}\mathcal{L}'$, we compute $\mu H(f, f')$ below:

$$\begin{aligned} & \mu H(f, f') \\ &= \text{fold}_{H^{(X,x)}} \left(\text{in}_{H^{(Y,y)}} \circ H \left((f, f'), \mu H^{(Y,y)} \right) \right) && \{ \text{Proposition 4} \} \\ &= \text{fold}_{H^{(X,x)}} \left(\left(\text{in}_{\overline{H}^Y}, \mathcal{L}(\text{in}_{\overline{H}^Y})(\text{in}_{h_Y^y}) \right) \circ H \left((f, f'), \mu H^{(Y,y)} \right) \right) && \{ \text{Eq. (33)} \} \\ &= \text{fold}_{H^{(X,x)}} \left(\left(\text{in}_{\overline{H}^Y}, \mathcal{L}(\text{in}_{\overline{H}^Y})(\text{in}_{h_Y^y}) \right) \circ \left(\overline{H}(f, \mu\overline{H}^Y), h_{(X, \mu\overline{H}^Y)}(f', \mu h_Y^y) \right) \right) && \{ \text{indexed functor} \} \\ &= \text{fold}_{H^{(X,x)}} \left(\text{in}_{\overline{H}^Y} \circ \overline{H}(f, \mu\overline{H}^Y), \mathcal{L} \left(\text{in}_{\overline{H}^Y} \circ \overline{H}(f, \mu\overline{H}^Y) \right) (\text{in}_{h_Y^y}) \right) \\ & \qquad \qquad \qquad \circ \left(h_{(X, \mu\overline{H}^Y)}(f', \mu h_Y^y) \right) && \{ \text{composing} \} \end{aligned}$$

which, by denoting $\xi = \text{in}_{\overline{H}^Y} \circ \overline{H}(f, \mu\overline{H}^Y)$ and $\xi' = \mathcal{L}(\xi) (\text{in}_{\underline{h}_Y^y}) \circ (h_{(X, \mu\overline{H}^Y)}(f', \mu\underline{h}_Y^y))$, is equal to

$$\begin{aligned} & \text{fold}_{H^{(X,x)}} \left(\text{in}_{\overline{H}^Y} \circ \overline{H}(f, \mu\overline{H}^Y), \xi' \right) \\ &= \left(\text{fold}_{\overline{H}^X} \left(\text{in}_{\overline{H}^Y} \circ \overline{H}(f, \mu\overline{H}^Y) \right), \left(\text{fold}_{\underline{h}_X^x} \mathcal{L} \left(\overline{H}^X(\text{fold}_{\overline{H}^X} \xi) \cdot \text{in}_{\overline{H}^X}^{-1} \right) (\xi') \right) \right) \quad \{ \text{Eq. (34)} \} \\ &= \left(\mu\overline{H}(f), \left(\text{fold}_{\underline{h}_X^x} \mathcal{L} \left(\overline{H}^X(\text{fold}_{\overline{H}^X} \xi) \cdot \text{in}_{\overline{H}^X}^{-1} \right) (\xi') \right) \right). \quad \{ \text{Proposition 4} \} \end{aligned}$$

The above shows that

$$\mu H(f, f') = \left(\mu\overline{H}(f), \left(\text{fold}_{\underline{h}_X^x} \mathcal{L} \left(\overline{H}^X(\text{fold}_{\overline{H}^X} \xi) \cdot \text{in}_{\overline{H}^X}^{-1} \right) (\xi') \right) \right). \tag{38}$$

Now, we can proceed to prove that μH is actually a split fibration functor. Firstly, by Eq. (38), we have that

$$\begin{array}{ccc} \Sigma_{\mathcal{D}} \mathcal{L}' & \xrightarrow{\mu H} & \Sigma_{\mathcal{C}} \mathcal{L} \\ \text{P}_{\mathcal{L}'} \downarrow & & \downarrow \text{P}_{\mathcal{L}} \\ \mathcal{D} & \xrightarrow{\mu \overline{H}} & \mathcal{C} \end{array} \tag{39}$$

commutes.

Let $(g, \text{id}) : (X, \mathcal{L}'(g)(y)) \rightarrow (Y, y)$ be a morphism in $(\Sigma_{\mathcal{D}} \mathcal{L}')$. Denoting, again,

$$\xi = \text{in}_{\overline{H}^Y} \circ \overline{H}(g, \mu\overline{H}^Y) \quad \text{and} \quad \xi' = \mathcal{L}(\xi) (\text{in}_{\underline{h}_Y^y}) \circ (h_{(X, \mu\overline{H}^Y)}(\text{id}, \mu\underline{h}_Y^y)),$$

we have that

$$\begin{aligned} & \left(\text{fold}_{\underline{h}_X^{X'}} \mathcal{L} \left(\overline{H}^X(\text{fold}_{\overline{H}^X} \xi) \cdot \text{in}_{\overline{H}^X}^{-1} \right) (\xi') \right) \\ &= \left(\text{fold}_{\underline{h}_X^{X'}} \mathcal{L} \left(\xi \cdot \overline{H}^X(\text{fold}_{\overline{H}^X} \xi) \cdot \text{in}_{\overline{H}^X}^{-1} \right) (\text{in}_{\underline{h}_Y^y}) \right) \quad \{ h_{(X, \mu\overline{H}^Y)}(\text{id}, \mu\underline{h}_Y^y) = \text{id} \} \\ &= \left(\text{fold}_{\underline{h}_X^{X'}} \mathcal{L} \left((\text{fold}_{\overline{H}^X} \xi) \cdot \text{in}_{\overline{H}^X} \cdot \text{in}_{\overline{H}^X}^{-1} \right) (\text{in}_{\underline{h}_Y^y}) \right) \quad \{ \text{fold}_{\overline{H}^X} \xi \} \\ &= \left(\text{fold}_{\underline{h}_X^{X'}} \mathcal{L} (\mu\overline{H}(g)) (\text{in}_{\underline{h}_Y^y}) \right) \quad \{ \text{Proposition 4} \} \\ &= \text{id}_{\mu\underline{h}_X^{X'} \mathcal{L}'(g)(y)} \quad \{ \text{Eq. (37)} \} \end{aligned}$$

By Eq. (38), the above proves that

$$\mu H(g, \text{id}) = (\mu\overline{H}(g), \text{id})$$

and, hence, we completed the proof that μH is a split fibration functor. □

We can, then, reformulate our result in terms of the existence of parameterized initial algebras in the base category and in the fibers. That is to say, we have:

Theorem 48 (Parameterized initial algebras are strictly indexed functors). *Let (\overline{H}, h) be a strictly indexed functor from $\mathcal{L}' \times \mathcal{L} : (\mathcal{D} \times \mathcal{C})^{\text{op}} \rightarrow \mathbf{Cat}$ to $\mathcal{L} : \mathcal{C}^{\text{op}} \rightarrow \mathbf{Cat}$, and $H : (\Sigma_{\mathcal{D}} \mathcal{L}') \times (\Sigma_{\mathcal{C}} \mathcal{L}) \rightarrow \Sigma_{\mathcal{C}} \mathcal{L}$ the corresponding split fibration functor. Assume that:*

- (h1) the parameterized initial algebra $\mu\overline{H} : \mathcal{D} \rightarrow \mathcal{C}$ exists;
- (h2) for any $X \in \mathcal{D}$, the parameterized initial algebra $\mu\underline{h}_X$ exists;

(h3) for each morphism $g : X \rightarrow Y$ in \mathcal{D} and $y \in Y$, Eq. (40) holds

$$\mathcal{L}(\mu\bar{H}(g))(\text{in}_{\underline{h}_Y}^y) = \text{in}_{\underline{h}_X}^{\mathcal{L}'(g)(y)} \tag{40}$$

In this setting, the parameterized initial algebra:

$$\mu H : \Sigma_{\mathcal{D}}\mathcal{L}' \rightarrow \Sigma_{\mathcal{C}}\mathcal{L}$$

is a split fibration functor coming from the strictly indexed functor $(\mu\bar{H}, \mu(\underline{h}_{(-)}))$ in which, for each $X \in \mathcal{D}$,

$$\mu(\underline{h}_{(X)}) = \mu\underline{h}_X = \mu\left(\mathcal{L}(\text{in}_{\bar{H}^X}^{-1})h_{(X, \mu\bar{H}^X)}\right) : \mathcal{L}'(X) \rightarrow \mathcal{L}(\mu\bar{H}^X). \tag{41}$$

Proof. By Theorem 47 (Eq. (38)) and Proposition 44 (Eq. (28)), we have that

$$\mu H : \Sigma_{\mathcal{D}}\mathcal{L}' \rightarrow \Sigma_{\mathcal{C}}\mathcal{L}$$

comes from the indexed category $(\mu\bar{H}, \mathfrak{h})$ in which, for each $X \in \mathcal{D}$ and each morphism $f' : x \rightarrow w$ in $\mathcal{L}'(X)$,

$$\begin{aligned} &\mathfrak{h}_X(f') \\ &= \mu H(\text{id}_X, f') \\ &= \left(\text{id}_{\mu\bar{H}^X}, \text{fold}_{\underline{h}_X} \left(\text{in}_{\underline{h}_X}^w \circ \mathcal{L} \left(\text{in}_{\bar{H}^X}^{-1}\right) \left(h_{(X, \mu\bar{H}^X)}(f', \mu\underline{h}_X^w)\right)\right)\right) \quad \{ \text{Eq. (38)} \} \\ &= \left(\text{id}_{\mu\bar{H}^X}, \text{fold}_{\underline{h}_X} \left(\text{in}_{\underline{h}_X}^w \circ \underline{h}_X(f', \mu\underline{h}_X^w)\right)\right) \\ &= \left(\text{id}_{\mu\bar{H}^X}, \mu\underline{h}_X(f')\right) \quad \{ \text{Proposition 4} \} \end{aligned}$$

□

Finally, for strictly indexed categories respecting initial algebras (see Definition 50), we get a cleaner version of Theorem 48 below.

Corollary 49 (Parameterized initial algebras and strictly indexed categories respecting initial algebras). *Let (\bar{H}, h) be a strictly indexed functor from $\mathcal{L}' \times \mathcal{L} : (\mathcal{D} \times \mathcal{C})^{\text{op}} \rightarrow \mathbf{Cat}$ to $\mathcal{L} : \mathcal{C}^{\text{op}} \rightarrow \mathbf{Cat}$ and $H : (\Sigma_{\mathcal{D}}\mathcal{L}') \times (\Sigma_{\mathcal{C}}\mathcal{L}) \rightarrow \Sigma_{\mathcal{C}}\mathcal{L}$ the corresponding split fibration functor. Assume that:*

- (h1) \mathcal{L} respects initial algebras;
- (h2) the parameterized initial algebra $\mu\bar{H} : \mathcal{D} \rightarrow \mathcal{C}$ exists;
- (h3) for any $X \in \mathcal{D}$, the parameterized initial algebra $\mu\underline{h}_X$ exists.

In this setting, the parameterized initial algebra:

$$\mu H : \Sigma_{\mathcal{D}}\mathcal{L}' \rightarrow \Sigma_{\mathcal{C}}\mathcal{L}$$

is a split fibration functor coming from the strictly indexed functor $(\mu\bar{H}, \mu(\underline{h}_{(-)}))$ in which, for each $X \in \mathcal{D}$,

$$\mu(\underline{h}_{(X)}) = \mu\underline{h}_X = \mu\left(\mathcal{L}(\text{in}_{\bar{H}^X}^{-1})h_{(X, \mu\bar{H}^X)}\right) : \mathcal{L}'(X) \rightarrow \mathcal{L}(\mu\bar{H}^X). \tag{42}$$

Proof. By Theorem 48, it is enough to show that Eq. (40) holds whenever \mathcal{L} respects initial algebras.

We have that, for any morphism $g : X \rightarrow Y$ in \mathcal{D} , and each $y \in \mathcal{L}'(Y)$, by the naturality of $h : \mathcal{L}' \times \mathcal{L} \rightarrow (\mathcal{L} \circ \overline{H}^{\text{op}})$ and the definition of $\mu \overline{H}(g)$, the squares

$$\begin{array}{ccc}
 \mathcal{L}(\mu \overline{H}^Y) & \xrightarrow{\mathcal{L}(\mu \overline{H}(g))} & \mathcal{L}(\mu \overline{H}^X) \\
 \downarrow (y, \text{id}_{\mathcal{L}(\mu \overline{H}^Y)}) & & \downarrow (\mathcal{L}'(y), \text{id}_{\mathcal{L}(\mu \overline{H}^X)}) \\
 \mathcal{L}'(Y) \times \mathcal{L}(\mu \overline{H}^Y) & \xrightarrow{\mathcal{L}'(g) \times \mathcal{L}(\mu \overline{H}(g))} & \mathcal{L}'(X) \times \mathcal{L}(\mu \overline{H}^X) \\
 \downarrow h_{(Y, \mu \overline{H}^Y)} & & \downarrow h_{(X, \mu \overline{H}^X)} \\
 \mathcal{L}(\overline{H}(Y, \mu \overline{H}^Y)) & \xrightarrow{\mathcal{L}(\overline{H}(g, \mu \overline{H}(g)))} & \mathcal{L}(\overline{H}(X, \mu \overline{H}^X)) \\
 \downarrow \mathcal{L}(\text{in}_{\overline{H}^Y})^{-1} & & \downarrow \mathcal{L}(\text{in}_{\overline{H}^X})^{-1} \\
 \mathcal{L}(\mu \overline{H}^Y) & \xrightarrow{\mathcal{L}(\mu \overline{H}(g))} & \mathcal{L}(\mu \overline{H}^X)
 \end{array}$$

commute. Thus, we get that

$$\begin{aligned}
 & \mathcal{L}(\mu \overline{H}(g)) \circ h_Y^y \\
 &= \mathcal{L}(\mu \overline{H}(g)) \circ h_Y \circ (y, \text{id}_{\mathcal{L}(\mu \overline{H}^Y)}) \\
 &= \mathcal{L}(\mu \overline{H}(g)) \circ \mathcal{L}(\text{in}_{\overline{H}^Y})^{-1} \circ h_{(Y, \mu \overline{H}^Y)} \circ (y, \text{id}_{\mathcal{L}(\mu \overline{H}^Y)}) \\
 &= \mathcal{L}(\text{in}_{\overline{H}^X})^{-1} \circ h_{(X, \mu \overline{H}^X)} \circ (\mathcal{L}'(y), \text{id}_{\mathcal{L}(\mu \overline{H}^X)}) \circ \mathcal{L}(\mu \overline{H}(g)) \\
 &= h_X^{\mathcal{L}'(y)} \circ \mathcal{L}(\mu \overline{H}(g)).
 \end{aligned}$$

Therefore, assuming that \mathcal{L} respects initial algebras, we conclude that

$$\mathcal{L}(\mu \overline{H}(g))(\text{in}_{h_Y^y}) = \text{in}_{h_X^{\mathcal{L}'(y)}}$$

holds. That is to say (40) holds for any $g : X \rightarrow Y$ in \mathcal{D} and any $y \in \mathcal{L}'(Y)$. This completes the proof by Theorem 49. □

6.11 General result on terminal coalgebras in total categories

Analogously to the case of initial algebras above, in order to give basis for our study in Section 6.12, we investigate the general case of parameterized terminal coalgebras of split fibration functors like in (32).

Definition 11 on initial algebra-preserving functors plays a central role in Theorem 51. Specifically, we use this definition in the context of indexed categories, where we define:

Definition 50 (Initial-algebra-respecting). *A strictly indexed category $\mathcal{L} : \mathcal{C}^{\text{op}} \rightarrow \mathbf{Cat}$ respects initial algebras if $\mathcal{L}(f)$ strictly preserves initial algebras for any morphism f of \mathcal{C} .⁵*

Dually, $\mathcal{L} : \mathcal{C}^{\text{op}} \rightarrow \mathbf{Cat}$ respects terminal coalgebras if $\mathcal{L}(f)$ strictly preserves terminal coalgebras for any morphism f of \mathcal{C} .

Theorem 51 (Terminal coalgebras of strictly indexed endofunctors). *Let (\overline{E}, e) be a strictly indexed endofunctor on $\mathcal{L} : \mathcal{C}^{\text{op}} \rightarrow \mathbf{Cat}$ and $E : \Sigma_{\mathcal{C}} \mathcal{L} \rightarrow \Sigma_{\mathcal{C}} \mathcal{L}$ the corresponding split fibration endofunctor. Assume that:*

- (e1) \mathcal{L} respects terminal coalgebras;
- (e2) the terminal \bar{E} -coalgebra $(\nu\bar{E}, \text{out}_{\bar{E}})$ exists;
- (e3) the terminal $(\mathcal{L}(\text{out}_{\bar{E}})e_{\nu\bar{E}})$ -coalgebra $(\nu(\mathcal{L}(\text{out}_{\bar{E}})e_{\nu\bar{E}}), \text{out}_{\mathcal{L}(\text{out}_{\bar{E}})e_{\nu\bar{E}}})$ exists.

Denoting by \bar{e} the endofunctor $\mathcal{L}(\text{out}_{\bar{E}})e_{\nu\bar{E}}$ on $\mathcal{L}(\nu\bar{E})$, the terminal E -coalgebra exists and is given by:

$$\nu E = (\nu\bar{E}, \nu\bar{e}), \quad \text{out}_E = (\text{out}_{\bar{E}}, \text{out}_{\bar{e}}). \tag{43}$$

Moreover, for each E -coalgebra,

$$((Y, y), (\xi, \xi') : (Y, y) \rightarrow E(Y, y)) = ((Y, y), (\xi : Y \rightarrow \bar{E}(Y), \xi' : y \rightarrow \mathcal{L}(\xi)e_Y(y))),$$

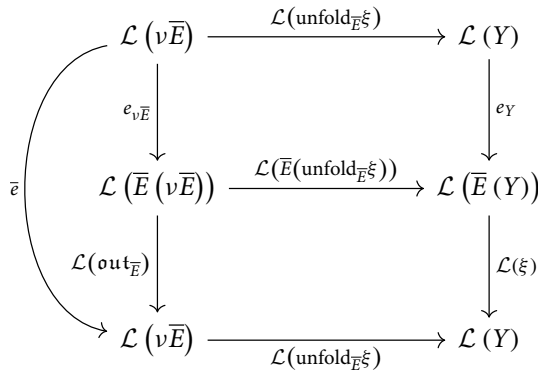
we have that

$$\text{unfold}_E(\xi, \xi') = (\text{unfold}_{\bar{E}}\xi, \text{unfold}_{\mathcal{L}(\xi)e_Y}\xi'). \tag{44}$$

Proof. Under the hypothesis above, given an E -coalgebra:

$$(\xi : Y \rightarrow \bar{E}(Y), \xi' : y \rightarrow \mathcal{L}(\xi)e_Y(y))$$

on (Y, y) , we have that the diagram:



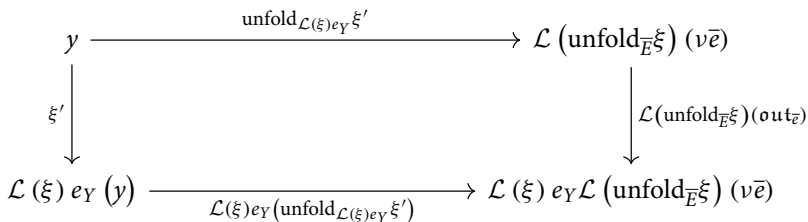
commutes. Thus, since \mathcal{L} respects terminal coalgebras, we have that

$$(\mathcal{L}(\text{unfold}_{\bar{E}}\xi)(\nu\bar{e}), \mathcal{L}(\text{unfold}_{\bar{E}}\xi)(\text{out}_{\bar{e}}))$$

is the terminal $\mathcal{L}(\xi)e_Y$ -coalgebra. Therefore, we have that

$$\text{unfold}_{\mathcal{L}(\xi)e_Y}\xi' : y \rightarrow \mathcal{L}(\text{unfold}_{\bar{E}}\xi)(\nu\bar{e})$$

is the unique morphism of $\mathcal{L}(Y)$ such that



which shows that

$$(\text{unfold}_{\bar{E}}\xi, \text{unfold}_{\mathcal{L}(\xi)e_Y}\xi') : (Y, y) \rightarrow E(Y, y) = (\bar{E}(Y), e_Y(y))$$

is the unique morphism of $\Sigma_C \mathcal{L}$ such that

$$\begin{array}{ccc}
 (Y, y) & \xrightarrow{(\text{unfold}_{\bar{E}\xi}, \text{unfold}_{\mathcal{L}(\xi)e_Y\xi'})} & (v\bar{E}, v\bar{e}) \\
 \downarrow (\xi, \xi') & & \downarrow (\text{out}_{\bar{E}}, \text{out}_{\bar{e}}) \\
 E(Y, y) = (\bar{E}(Y), e_Y(y)) & \xrightarrow{\frac{(\bar{E}(\text{unfold}_{\bar{E}\xi}), e_Y(\text{unfold}_{\mathcal{L}(\xi)e_Y\xi'}))}{E(\text{unfold}_{\bar{E}\xi}, \text{unfold}_{\mathcal{L}(\xi)e_Y\xi'})}} & E(v\bar{E}, v\bar{e}) = (\bar{E}(v\bar{E}), \bar{e}(v\bar{e}))
 \end{array}$$

commutes. This completes the proof that $vE = (v\bar{E}, v\bar{e})$ is the terminal E -coalgebra. □

Theorem 52 (Parameterized terminal coalgebras are strictly indexed functors). *Let (\bar{H}, h) be a strictly indexed functor from $\mathcal{L}' \times \mathcal{L} : (\mathcal{D} \times \mathcal{C})^{\text{op}} \rightarrow \mathbf{Cat}$ to $\mathcal{L} : \mathcal{C}^{\text{op}} \rightarrow \mathbf{Cat}$, and $H : (\Sigma_{\mathcal{D}}\mathcal{L}') \times (\Sigma_{\mathcal{C}}\mathcal{L}) \rightarrow \Sigma_{\mathcal{C}}\mathcal{L}$ the corresponding split fibration functor. Assume that*

- (h1) \mathcal{L} respects terminal coalgebras;
- (h2) for each object X of \mathcal{C} , the terminal \bar{H}^X -coalgebra $(v\bar{H}^X, \text{out}_{\bar{H}^X})$ exists;
- (h3) for each object (X, x) in $\Sigma_{\mathcal{D}}\mathcal{L}'$, denoting by \bar{h}_X the functor:

$$\mathcal{L}(\text{out}_{\bar{H}^X})h_{(X, v\bar{H}^X)} : \mathcal{L}'(X) \times \mathcal{L}(v\bar{H}^X) \rightarrow \mathcal{L}(v\bar{H}^X) \tag{45}$$

is such that the terminal \bar{h}_X -coalgebra $(v\bar{h}_X, \text{out}_{\bar{h}_X})$ exists.

In this setting, the parameterized terminal coalgebra:

$$vH : \Sigma_{\mathcal{D}}\mathcal{L}' \rightarrow \Sigma_{\mathcal{C}}\mathcal{L}$$

is a split fibration functor coming from the strictly indexed functor $(v\bar{H}, v(\bar{h}_{(-)}))$ in which, for each $X \in \mathcal{D}$,

$$v(\bar{h}_{(X)}) = v\bar{h}_X = v\left(\mathcal{L}(\text{out}_{\bar{H}^X})h_{(X, v\bar{H}^X)}\right) : \mathcal{L}'(X) \rightarrow \mathcal{L}(v\bar{H}^X). \tag{46}$$

Proof. Assuming the hypothesis, we conclude that, for each (X, x) in $\Sigma_{\mathcal{D}}\mathcal{L}'$, $\Sigma_{\mathcal{C}}\mathcal{L}$ has the terminal $H^{(X,x)}$ -coalgebra by Theorem 51. Hence, by Proposition 4, we have that

$$vH : \Sigma_{\mathcal{D}}\mathcal{L}' \rightarrow \Sigma_{\mathcal{C}}\mathcal{L}$$

exists. More precisely, given a morphism $(f, f') : (X, x) \rightarrow (Y, y)$ in $\Sigma_{\mathcal{D}}\mathcal{L}'$, we compute $vH(f, f')$ below:

$$\begin{aligned}
 & vH(f, f') \\
 &= \text{unfold}_{H(Y,y)} \left(H \left((f, f'), vH^{(X,x)} \right) \circ \text{out}_{H^{(X,x)}} \right) && \{ \text{Proposition 4} \} \\
 &= \text{unfold}_{H(Y,y)} \left(H \left((f, f'), vH^{(X,x)} \right) \circ \left(\text{out}_{\bar{H}^X}, \text{out}_{\bar{h}_X} \right) \right) && \{ \text{Eq. (43)} \} \\
 &= \text{unfold}_{H(Y,y)} \left(\left(\bar{H}(f, v\bar{H}^X), h_{(X, v\bar{H}^X)}(f', v\bar{h}_X^x) \right) \circ \left(\text{out}_{\bar{H}^X}, \text{out}_{\bar{h}_X} \right) \right) && \{ \text{hypothesis} \} \\
 &= \text{unfold}_{H(Y,y)} \left(\bar{H}(f, v\bar{H}^X) \circ \text{out}_{\bar{H}^X}, \mathcal{L} \left(\text{out}_{\bar{H}^X} \right) \left(h_{(X, v\bar{H}^X)}(f', v\bar{h}_X^x) \right) \circ \text{out}_{\bar{h}_X} \right) && \{ \text{composing} \} \\
 &= \text{unfold}_{H(Y,y)} \left(\bar{H}(f, v\bar{H}^X) \circ \text{out}_{\bar{H}^X}, \bar{h}_X \left(f', v\bar{h}_X^x \right) \circ \text{out}_{\bar{h}_X} \right) && \{ \text{definition of } \bar{h}_X \}
 \end{aligned}$$

$$\begin{aligned}
 &= \left(\text{unfold}_{\overline{H}^Y} \left(\overline{H}(f, \nu \overline{H}^X) \circ \text{out}_{\overline{H}^X} \right), \text{unfold}_{\overline{h}_Y} \left(\overline{h}_X \left(f', \nu \overline{h}_X^x \right) \circ \text{out}_{\overline{h}_X^x} \right) \right) && \{ \text{Eq. (44)} \} \\
 &= \left(\nu \overline{H}(f), \nu \overline{h}_Y(f') \right) && \{ \text{Proposition 4} \}
 \end{aligned}$$

Since $\nu H(f, f') = (\nu \overline{H}(f), \nu \overline{h}_Y(f'))$, clearly, then, the pair $(\nu H, \nu \overline{H})$ satisfies Eqs. (26) and (27) of Proposition 26. Moreover, νH comes from the strictly indexed functor $(\nu \overline{H}, \nu (\overline{h}_{(-)}))$. \square

6.12 $\mu\nu$ -polynomials in total categories

We examine the existence of $\mu\nu$ -polynomials in $\Sigma_{\mathcal{C}}\mathcal{L}$ and $\Sigma_{\mathcal{C}}\mathcal{L}^{\text{op}}$. In order to do so, we employ the results and terminology established in Theorem 46 and Section 6.11

Making use of Definitions 31 and 19, we introduce the following concept to provide support for our definition of Σ -bimodel for inductive and coinductive types:

Definition 53 ($\mu\nu\text{Poly}_{\mathcal{L}}$). *Let \mathcal{C} be a category with $\mu\nu$ -polynomials, and $\mathcal{L} : \mathcal{C}^{\text{op}} \rightarrow \mathbf{Cat}$ an extensive strictly indexed category with strictly indexed finite biproducts. We define the category $\mu\nu\text{Poly}_{\mathcal{L}}$ as the smallest subcategory of \mathbf{Cat} satisfying the following.*

- (O) *The objects are defined inductively by:*
 - (O1) *the terminal category $\mathbb{1}$ is an object of $\mu\nu\text{Poly}_{\mathcal{L}}$;*
 - (O2) *if \mathcal{D} and \mathcal{D}' are objects of $\mu\nu\text{Poly}_{\mathcal{L}}$, then so is $\mathcal{D} \times \mathcal{D}'$;*
 - (O3) *for each object $W \in \mathcal{C}$, the category $\mathcal{L}(W)$ is an object of $\mu\nu\text{Poly}_{\mathcal{L}}$.*
- (M) *The morphisms satisfy the following properties:*
 - (M1) *for any object \mathcal{D} of $\mu\nu\text{Poly}_{\mathcal{L}}$, the unique functor $\mathcal{D} \rightarrow \mathbb{1}$ is a morphism of $\mu\nu\text{Poly}_{\mathcal{L}}$;*
 - (M2) *for any object \mathcal{D} of $\mu\nu\text{Poly}_{\mathcal{L}}$, all the functors $\mathbb{1} \rightarrow \mathcal{D}$ are morphisms of $\mu\nu\text{Poly}_{\mathcal{L}}$;*
 - (M3) *for each $(W, X) \in \mathcal{C} \times \mathcal{C}$, the projections $\pi_1 : \mathcal{D} \times \mathcal{D}' \rightarrow \mathcal{D}$ and $\pi_2 : \mathcal{D} \times \mathcal{D}' \rightarrow \mathcal{D}'$ are morphisms of $\mu\nu\text{Poly}_{\mathcal{L}}$;*
 - (M4) *for each $W \in \mathcal{C}$, the biproduct $+: \mathcal{L}(W) \times \mathcal{L}(W) \rightarrow \mathcal{L}(W)$ is a morphism of $\mu\nu\text{Poly}_{\mathcal{L}}$;*
 - (M5) *for each $(W, X) \in \mathcal{C} \times \mathcal{C}$, the functor:*

$$S^{(W,X)} : \mathcal{L}(W) \times \mathcal{L}(X) \rightarrow \mathcal{L}(W \sqcup X)$$

- of the extensive structure (see (11)) is a morphism of $\mu\nu\text{Poly}_{\mathcal{L}}$;*
- (M6) *given an object \mathcal{D} of $\mu\nu\text{Poly}_{\mathcal{C}}$, a morphism $\overline{H} : \mathcal{D} \times \mathcal{C} \rightarrow \mathcal{C}$ of $\mu\nu\text{Poly}_{\mathcal{C}}$ and any object $X \in \mathcal{D}'$,*

$$\begin{aligned}
 \mathcal{L}(\text{in}_{\overline{H}^X})^{-1} &: \mathcal{L} \left(\overline{H}^X \left(\mu \overline{H}^X \right) \right) \rightarrow \mathcal{L} \left(\mu \overline{H}^X \right), \\
 \mathcal{L}(\text{out}_{\overline{H}^X}) &: \mathcal{L} \left(\overline{H}^X \left(\nu \overline{H}^X \right) \right) \rightarrow \mathcal{L} \left(\nu \overline{H}^X \right)
 \end{aligned}$$

- are morphisms of $\mu\nu\text{Poly}_{\mathcal{L}}$;*
- (M7) *for each $(W, X) \in \mathcal{C} \times \mathcal{C}$, the functors induced by the projections:*

$$\mathcal{L}(\pi_1) : \mathcal{L}(W) \rightarrow \mathcal{L}(W \times X), \quad \mathcal{L}(\pi_2) : \mathcal{L}(X) \rightarrow \mathcal{L}(W \times X)$$

- are morphisms of $\mu\nu\text{Poly}_{\mathcal{L}}$;*
- (M8) *if $E : \mathcal{D} \rightarrow \mathcal{D}'$ and $J : \mathcal{D} \rightarrow \mathcal{D}''$ are morphisms of $\mu\nu\text{Poly}_{\mathcal{L}}$, then so is $(E, J) : \mathcal{D} \rightarrow \mathcal{D}' \times \mathcal{D}''$;*
- (M9) *if $\mathcal{D}', \mathcal{D}$ are objects of $\mu\nu\text{Poly}_{\mathcal{L}}$, $h : \mathcal{D}' \times \mathcal{D} \rightarrow \mathcal{D}$ is a morphism of $\mu\nu\text{Poly}_{\mathcal{L}}$ and $\mu h : \mathcal{D}' \rightarrow \mathcal{D}$ exists, then μh is a morphism of $\mu\nu\text{Poly}_{\mathcal{L}}$;*
- (M10) *if $\mathcal{D}', \mathcal{D}$ are objects of $\mu\nu\text{Poly}_{\mathcal{L}}$, $h : \mathcal{D}' \times \mathcal{D} \rightarrow \mathcal{D}$ is a morphism of $\mu\nu\text{Poly}_{\mathcal{L}}$ and $\nu h : \mathcal{D}' \rightarrow \mathcal{D}$ exists, then νh is a morphism of $\mu\nu\text{Poly}_{\mathcal{L}}$.*

Having established the previous definition, we can now introduce the notion of a Σ -bimodel for inductive and coinductive types:

Definition 54 (Σ -bimodel for inductive and coinductive types). *We say that $\mathcal{L} : \mathcal{C}^{\text{op}} \rightarrow \mathbf{Cat}$ is a Σ -bimodel for inductive and coinductive types (or, for short, a $*$ -indexed category) if:*

- (*1) \mathcal{L} is a strictly indexed category;
- (*2) \mathcal{C} has $\mu\nu$ -polynomials (Definition 6);
- (*3) $\mathcal{L} : \mathcal{C}^{\text{op}} \rightarrow \mathbf{Cat}$ has strictly indexed finite biproducts (Definition 19);
- (*4) \mathcal{L} is extensive (Definition 31);
- (*5) \mathcal{L} respects terminal coalgebras and initial algebras (Definition 50);
- (*6) whenever \mathcal{D} is an object of $\mu\nu\text{Poly}_{\mathcal{L}}$ and $e : \mathcal{D} \rightarrow \mathcal{D}$ is a morphism of $\mu\nu\text{Poly}_{\mathcal{L}}$, μe and νe exist.

Lemma 55. *Let $\mathcal{L} : \mathcal{C}^{\text{op}} \rightarrow \mathbf{Cat}$ be a $*$ -indexed category. If $\mathcal{D}, \mathcal{D}'$ are objects of $\mu\nu\text{Poly}_{\mathcal{L}}$ then, whenever $h : \mathcal{D}' \times \mathcal{D} \rightarrow \mathcal{D}$ is a morphism of $\mu\nu\text{Poly}_{\mathcal{L}}$,*

$$\mu h : \mathcal{D}' \rightarrow \mathcal{D} \quad \text{and} \quad \nu h : \mathcal{D}' \rightarrow \mathcal{D}$$

exist.

Proof. By Proposition 4, it is enough to show that, for each $x \in \mathcal{D}'$, μh^x and νh^x exist.

In fact, denoting by $x : \mathbb{1} \rightarrow \mathcal{D}'$ the functor constantly equal to $x \in \mathcal{D}'$, the functor h^x is the composition below:

$$\begin{array}{ccccccc} \mathcal{D} & \xrightarrow{(1, \text{id}_{\mathcal{D}})} & \mathbb{1} \times \mathcal{D} & \xrightarrow{(x \circ \pi_1, \text{id}_{\mathcal{D}} \circ \pi_2)} & \mathcal{D}' \times \mathcal{D} & \xrightarrow{h} & \mathcal{D} \\ & & & & & \searrow & \\ & & & & & & h^x \end{array}$$

Since all the horizontal arrows above are morphisms of $\mu\nu\text{Poly}_{\mathcal{L}}$, we conclude that h^x is an endomorphism of $\mu\nu\text{Poly}_{\mathcal{L}}$. Therefore, since \mathcal{L} is a $*$ -indexed category, μh^x and νh^x exist. \square

Definition 56 ($\mu\nu\mathcal{L}$ -indexed category and indexed functor). *Let $\mathcal{L} : \mathcal{C}^{\text{op}} \rightarrow \mathbf{Cat}$, $\mathcal{L}' : \mathcal{D}^{\text{op}} \rightarrow \mathbf{Cat}$ be strictly indexed categories. We say that \mathcal{L}' is a $\mu\nu\mathcal{L}$ -indexed category if:*

- ($\mu\nu\mathcal{L}1$) \mathcal{D} is an object of $\mu\nu\text{Poly}_{\mathcal{L}}$;
- ($\mu\nu\mathcal{L}2$) $\mathcal{L}'(W)$ is an object of $\mu\nu\text{Poly}_{\mathcal{L}}$ for any W in \mathcal{D} .

A strictly indexed functor (\overline{H}, h) between $\mathcal{L}' : \mathcal{D}^{\text{op}} \rightarrow \mathbf{Cat}$ and $\mathcal{L}'' : \mathcal{E}^{\text{op}} \rightarrow \mathbf{Cat}$ is a $\mu\nu\mathcal{L}$ -indexed functor if:

- ($\mu\nu\mathcal{L}3$) $\mathcal{L}', \mathcal{L}''$ are $\mu\nu\mathcal{L}$ -indexed categories;
- ($\mu\nu\mathcal{L}4$) $\overline{H} : \mathcal{D} \rightarrow \mathcal{E}$ is a morphism of $\mu\nu\text{Poly}_{\mathcal{L}}$;
- ($\mu\nu\mathcal{L}5$) for each $X \in \mathcal{D}$, $h_X : \mathcal{L}'(X) \rightarrow \mathcal{L}'' \circ \overline{H}(X)$ is a morphism of $\mu\nu\text{Poly}_{\mathcal{L}}$.

Theorem 57. *Let $\mathcal{L}' : \mathcal{D}^{\text{op}} \rightarrow \mathbf{Cat}$ be a strictly indexed category and $\mathcal{L} : \mathcal{C}^{\text{op}} \rightarrow \mathbf{Cat}$ a $*$ -indexed category. Assume that (\overline{H}, h) is a $\mu\nu\mathcal{L}$ -indexed functor, and $H : \Sigma_{\mathcal{E} \times \mathcal{D}} (\mathcal{L}' \times \mathcal{L}) \cong (\Sigma_{\mathcal{E}} \mathcal{L}') \times (\Sigma_{\mathcal{D}} \mathcal{L}) \rightarrow \Sigma_{\mathcal{D}} \mathcal{L}$ is the corresponding split fibration functor. We have that:*

(i) $\mu H : \Sigma_{\mathcal{E}}\mathcal{L}' \rightarrow \Sigma_{\mathcal{D}}\mathcal{L}$ exists and is the split fibration functor induced by the $\mu\nu\mathcal{L}$ -indexed functor:

$$(\mu\bar{H} : \mathcal{E} \rightarrow \mathcal{D}, \mu(\underline{h}_{(-)})) \tag{47}$$

in which

$$\mu(\underline{h}_{(X)}) = \mu h_X = \mu \left(\mathcal{L}(\text{in}_{\bar{H}^X})^{-1} h_{(X, \mu\bar{H}^X)} \right) : \mathcal{L}'(X) \rightarrow \mathcal{L}(\mu\bar{H}^X). \tag{48}$$

(ii) $\nu H : \Sigma_{\mathcal{E}}\mathcal{L}' \rightarrow \Sigma_{\mathcal{D}}\mathcal{L}$ exists and is the split fibration functor induced by the $\mu\nu\mathcal{L}$ -indexed functor:

$$(\nu\bar{H} : \mathcal{E} \rightarrow \mathcal{D}, \nu(\bar{h}_{(-)})) \tag{49}$$

in which

$$\nu(\bar{h}_{(X)}) = \nu \bar{h}_X = \nu \left(\mathcal{L}(\text{out}_{\bar{H}^X}) h_{(X, \nu\bar{H}^X)} \right) : \mathcal{L}''(X) \rightarrow \mathcal{L}'(\nu\bar{H}^X). \tag{50}$$

Furthermore, both μH and νH are $\mu\nu\mathcal{L}$ -indexed functors.

Proof. Since \mathcal{C} has $\mu\nu$ -polynomials, \mathcal{D} is an object of $\mu\nu\text{Poly}_{\mathcal{C}}$ and \bar{H} is a morphism of $\mu\nu\text{Poly}_{\mathcal{C}}$, we have that $\mu\bar{H}$ and $\nu\bar{H}$ exist by Lemma 8 (and, hence, are morphisms in $\mu\nu\text{Poly}_{\mathcal{C}}$). Moreover, we have that $\mathcal{L}(\text{out}_{\bar{H}^X})$ and $\mathcal{L}(\text{in}_{\bar{H}^X})^{-1}$ are morphisms of $\mu\nu\text{Poly}_{\mathcal{C}}$ by (M6) of Definition 53.

For any $X \in \mathcal{D}$, since (\bar{H}, h) is a $\mu\nu\mathcal{L}$ -indexed functor, we have that, $\mathcal{L}'(X)$ is an object of $\mu\nu\text{Poly}_{\mathcal{L}}$ and

$$\begin{aligned} h_{(X, \mu\bar{H}^X)} : \mathcal{L}'(X) \times \mathcal{L}(\mu\bar{H}^X) &\rightarrow \mathcal{L} \circ \bar{H}(X, \mu\bar{H}^X) \\ h_{(X, \nu\bar{H}^X)} : \mathcal{L}'(X) \times \mathcal{L}(\nu\bar{H}^X) &\rightarrow \mathcal{L} \circ \bar{H}(X, \nu\bar{H}^X) \end{aligned}$$

are morphisms of $\mu\nu\text{Poly}_{\mathcal{L}}$.

We conclude, then, that the compositions:

$$\begin{aligned} \underline{h}_X = \mathcal{L}(\text{in}_{\bar{H}^X})^{-1} h_{(X, \mu\bar{H}^X)} : \mathcal{L}'(X) \times \mathcal{L}(\mu\bar{H}^X) &\rightarrow \mathcal{L}(\mu\bar{H}^X) \\ \bar{h}_X = \mathcal{L}(\text{out}_{\bar{H}^X}) h_{(X, \nu\bar{H}^X)} : \mathcal{L}'(X) \times \mathcal{L}(\nu\bar{H}^X) &\rightarrow \mathcal{L}(\nu\bar{H}^X) \end{aligned}$$

are also morphisms of $\mu\nu\text{Poly}_{\mathcal{L}}$. Thus, we have that $\mu\underline{h}_X$ and $\nu\bar{h}_X$ exist (and are morphisms of $\mu\nu\text{Poly}_{\mathcal{L}}$) by Lemma 55.

Finally, since \mathcal{L} respects initial algebras and terminal coalgebras, we have that (\bar{H}, h) satisfies the hypotheses of Corollary 49 and Theorem 52. Therefore, μH and νH exist and are induced by (47) and (49), respectively.

The fact that (47) and (49) are also $\mu\nu\mathcal{L}$ -indexed functors follows from the fact that \mathcal{L}' is a $\mu\nu\mathcal{L}$ -indexed category by hypothesis, $\mu\bar{H}$ is a morphism of $\mu\nu\text{Poly}_{\mathcal{C}}$ (as observed above), and $\mu\underline{h}_X, \nu\bar{h}_X$ are morphisms of $\mu\nu\text{Poly}_{\mathcal{L}}$ (also observed above). □

In particular, we see that initial algebras and terminal coalgebras of $\mu\nu$ -polynomials in $\Sigma_{\mathcal{C}}\mathcal{L}$ (and, codually, $\Sigma_{\mathcal{C}}\mathcal{L}^{op}$) are fibred over \mathcal{C} .

Before proving Theorem 63, our main theorem about $\mu\nu$ -polynomials in $\Sigma_{\mathcal{C}}\mathcal{L}$, we prove Lemma 60 which establishes a bijection between objects of $\mu\nu\text{Poly}_{\Sigma_{\mathcal{C}}\mathcal{L}}$ and indexed categories.

Definition 58. Let $\mathcal{L} : \mathcal{C}^{\text{op}} \rightarrow \mathbf{Cat}$ be a strictly indexed category. We inductively define the set $\overline{\underline{\times}} \mathcal{L}$ of indexed categories as follows:

- $\overline{\underline{\times}} \mathcal{L}1$. the terminal indexed category $\mathbb{1} : \mathbb{1} \rightarrow \mathbf{Cat}$ belongs to $\overline{\underline{\times}} \mathcal{L}$;
- $\overline{\underline{\times}} \mathcal{L}2$. \mathcal{L} belongs to $\overline{\underline{\times}} \mathcal{L}$;
- $\overline{\underline{\times}} \mathcal{L}3$. if \mathcal{L}' and \mathcal{L}'' belong to $\overline{\underline{\times}} \mathcal{L}$, then $(\mathcal{L}' \underline{\times} \mathcal{L}'') \in \overline{\underline{\times}} \mathcal{L}$.

Lemma 59. Let $\mathcal{L} : \mathcal{C}^{\text{op}} \rightarrow \mathbf{Cat}$ be a strictly indexed category. Then all the elements of $\overline{\underline{\times}} \mathcal{L}$ are $\mu\nu\mathcal{L}$ -indexed categories.

Proof. The terminal indexed category $\mathbb{1} : \mathbb{1} \rightarrow \mathbf{Cat}$ is a $\mu\nu\mathcal{L}$ -indexed category since $\mathbb{1} \in \mu\nu\text{Poly}_{\mathcal{C}}$ and $\mathbb{1} \in \mu\nu\text{Poly}_{\mathcal{L}}$. Furthermore, $\mathcal{L} : \mathcal{C}^{\text{op}} \rightarrow \mathbf{Cat}$ is a $\mu\nu\mathcal{L}$ -indexed category by the definition of $\mu\nu\text{Poly}_{\mathcal{L}}$.

Finally, if $\mathcal{L}' : \mathcal{D}^{\text{op}} \rightarrow \mathbf{Cat}$ and $\mathcal{L}'' : \mathcal{E}^{\text{op}} \rightarrow \mathbf{Cat}$ are $\mu\nu\mathcal{L}$ -indexed categories, then:

– we have that $(\mathcal{D}, \mathcal{E}) \in \mu\nu\text{Poly}_{\mathcal{C}} \times \mu\nu\text{Poly}_{\mathcal{C}}$. Thus,

$$(\mathcal{D} \times \mathcal{E}) \in \mu\nu\text{Poly}_{\mathcal{C}}; \tag{51}$$

– for any $(W, W') \in \mathcal{D} \times \mathcal{E}$, the categories $\mathcal{L}'(W)$ and $\mathcal{L}''(W')$ are objects of $\mu\nu\text{Poly}_{\mathcal{L}}$. Thus,

$$\mathcal{L}' \underline{\times} \mathcal{L}'' (W, W') = \mathcal{L}'(W) \times \mathcal{L}''(W') \in \mu\nu\text{Poly}_{\mathcal{L}}. \tag{52}$$

By (51) and (52), we conclude that $\mathcal{L}' \underline{\times} \mathcal{L}'' : (\mathcal{D} \times \mathcal{E})^{\text{op}} \rightarrow \mathbf{Cat}$ is a $\mu\nu\mathcal{L}$ -indexed category. □

Lemma 60. Let $\mathcal{L} : \mathcal{C}^{\text{op}} \rightarrow \mathbf{Cat}$ be a strictly indexed category. The function

$$\bar{\partial} : \text{obj}(\mu\nu\text{Poly}_{\Sigma_{\mathcal{C}}\mathcal{L}}) \rightarrow \overline{\underline{\times}} \mathcal{L} \tag{53}$$

inductively defined by $\bar{\partial}1$, $\bar{\partial}2$, and $\bar{\partial}3$ is a bijection.

- $\bar{\partial}1$. terminal respecting: $\bar{\partial}(\mathbb{1}) := (\mathbb{1} : \mathbb{1} \rightarrow \mathbf{Cat})$;
- $\bar{\partial}2$. basic element: $\bar{\partial}(\Sigma_{\mathcal{C}}\mathcal{L}) := (\mathcal{L} : \mathcal{C}^{\text{op}} \rightarrow \mathbf{Cat})$;
- $\bar{\partial}3$. product respecting: given $(\mathcal{D}, \mathcal{D}') \in \mu\nu\text{Poly}_{\Sigma_{\mathcal{C}}\mathcal{L}} \times \mu\nu\text{Poly}_{\Sigma_{\mathcal{C}}\mathcal{L}}$,

$$\bar{\partial}(\mathcal{D} \times \mathcal{D}') := \bar{\partial}(\mathcal{D}) \underline{\times} \bar{\partial}(\mathcal{D}').$$

Proof. The inverse of $\bar{\partial}$ is clearly given by the Grothendieck construction. More precisely, the inverse is denoted herein by $\bar{\Sigma}$ and can be inductively defined as follows:

- $(\bar{\Sigma}1)$ terminal respecting: $\bar{\Sigma}(\mathbb{1} : \mathbb{1} \rightarrow \mathbf{Cat}) := \mathbb{1}$;
- $(\bar{\Sigma}2)$ basic element: $\bar{\Sigma}(\mathcal{L} : \mathcal{C}^{\text{op}} \rightarrow \mathbf{Cat}) := \Sigma_{\mathcal{C}}\mathcal{L}$;
- $(\bar{\Sigma}3)$ product respecting: given $(\mathcal{L}' : \mathcal{D}^{\text{op}} \rightarrow \mathbf{Cat}, \mathcal{L}'' : \mathcal{E}^{\text{op}} \rightarrow \mathbf{Cat}) \in \overline{\underline{\times}} \mathcal{L} \times \overline{\underline{\times}} \mathcal{L}$,

$$\bar{\Sigma}(\mathcal{L}' \underline{\times} \mathcal{L}'') := \bar{\Sigma}(\mathcal{L}') \times \bar{\Sigma}(\mathcal{L}'').$$

By the inductive definitions of the sets $\text{obj}(\mu\nu\text{Poly}_{\Sigma_{\mathcal{C}}\mathcal{L}})$ and $\overline{\underline{\times}} \mathcal{L}$, we conclude that

$$\bar{\Sigma} \circ \bar{\partial} = \text{id}_{\text{obj}(\mu\nu\text{Poly}_{\Sigma_{\mathcal{C}}\mathcal{L}})} \quad \text{and} \quad \bar{\partial} \circ \bar{\Sigma} = \text{id}_{\overline{\underline{\times}} \mathcal{L}}.$$

□

Lemma 61. *Let $\mathcal{L} : \mathcal{C}^{\text{op}} \rightarrow \mathbf{Cat}$ be a strictly indexed category. The objects of $\mu\nu\text{Poly}_{\Sigma_C \mathcal{L}}$ with the functors that are induced by $\mu\nu\mathcal{L}$ -indexed functors between objects of $\overline{\mathcal{X}}\mathcal{L}$ form a subcategory of \mathbf{Cat} .*

Proof. Let \mathcal{A} be an object of $\mu\nu\text{Poly}_{\Sigma_C \mathcal{L}}$. By Lemma 60, we have the associated strictly indexed category:

$$\overline{\partial}(\mathcal{A}) = \mathcal{L}' : \mathcal{D}^{\text{op}} \rightarrow \mathbf{Cat}.$$

The identity $\text{id}_{\mathcal{A}}$ on \mathcal{A} clearly comes from the identity:

$$(\text{id}_{\mathcal{D}} : \mathcal{D} \rightarrow \mathcal{D}, \text{id}) : \mathcal{L}' \rightarrow \mathcal{L}'$$

which is a $\mu\nu\mathcal{L}$ -indexed category, since \mathcal{L}' is a $\mu\nu\mathcal{L}$ -indexed category by Lemma 59.

Finally, if $E : \mathcal{A} \rightarrow \mathcal{A}'$ and $H : \mathcal{A}' \rightarrow \mathcal{A}''$ are functors induced, respectively, by the $\mu\nu\mathcal{L}$ -indexed functors:

$$(\overline{E}, e) : \mathcal{L}' \rightarrow \mathcal{L}'' \quad \text{and} \quad (\overline{H}, h) : \mathcal{L}'' \rightarrow \mathcal{L}''',$$

then $H \circ E$ is induced by the composition:

$$(\overline{H} \circ \overline{E}, h_{\overline{E}^{\text{op}}} \circ e)$$

which is a $\mu\nu\mathcal{L}$ -indexed functor as well, since $\overline{H}, \overline{E}$ are morphisms of $\mu\nu\text{Poly}_{\mathcal{C}}$ and, for any $W \in \mathcal{D}$, $h_{\overline{E}(W)}$ and e_W are morphisms of $\mu\nu\text{Poly}_{\mathcal{L}}$. □

Definition 62. *We denote by $\overline{\mu\nu\text{Poly}_{\Sigma_C \mathcal{L}}}$ the category defined in Lemma 61.*

Theorem 63. *Let $\mathcal{L} : \mathcal{C}^{\text{op}} \rightarrow \mathbf{Cat}$ be a $*$ -indexed category. The category $\Sigma_C \mathcal{L}$ has $\mu\nu$ -polynomials.*

Proof. By Theorem 57, since \mathcal{L} is a $*$ -indexed category, any endomorphism $E : \Sigma_C \mathcal{L} \rightarrow \Sigma_C \mathcal{L}$ of the subcategory $\overline{\mu\nu\text{Poly}_{\Sigma_C \mathcal{L}}}$ has an initial algebra and a terminal coalgebra. Therefore, in order to complete the proof, it is enough to show that the morphisms of $\overline{\mu\nu\text{Poly}_{\Sigma_C \mathcal{L}}}$ satisfy the inductive properties of Definition 6.

Let $\mathcal{A}, \mathcal{A}'$, and \mathcal{A}'' be objects of $\mu\nu\text{Poly}_{\Sigma_C \mathcal{L}}$. By Lemma 60, we have the associated strictly indexed categories:

$$\begin{aligned} \overline{\partial}(\mathcal{A}) &= \mathcal{L}' : \mathcal{D}^{\text{op}} \rightarrow \mathbf{Cat}, \\ \overline{\partial}(\mathcal{A}') &= \mathcal{L}'' : \mathcal{E}^{\text{op}} \rightarrow \mathbf{Cat}, \\ \overline{\partial}(\mathcal{A}'') &= \mathcal{L}''' : \mathcal{F}^{\text{op}} \rightarrow \mathbf{Cat}. \end{aligned}$$

Recall that $\mathcal{L}', \mathcal{L}''$ and \mathcal{L}''' are $\mu\nu\mathcal{L}$ -indexed categories by Lemma 59.

(A) The unique functor $\mathcal{A} \rightarrow \mathbb{1}$ is induced by the unique indexed functor:

$$(\mathcal{D} \rightarrow \mathbb{1}, (\mathcal{L}'(W) \rightarrow \mathbb{1})_{W \in \mathcal{D}})$$

between \mathcal{L} and the terminal indexed category $\mathbb{1} : \mathbb{1} \rightarrow \mathbf{Cat}$. Since $\mathcal{D} \rightarrow \mathbb{1}$ is a morphism of $\mu\nu\text{Poly}_{\mathcal{C}}$ and, for any $W \in \mathcal{D}$, $\mathcal{L}'(W) \rightarrow \mathbb{1}$ is a morphism of $\mu\nu\text{Poly}_{\mathcal{L}}$, we have that the unique indexed functor is a $\mu\nu\mathcal{L}$ -indexed functor.

(B) Given a functor $F : \mathbb{1} \rightarrow \mathcal{A} \cong \Sigma_C \mathcal{L}'$, it corresponds to an object $(W \in \mathcal{D}, x \in \mathcal{L}'(W)) \in \Sigma_C \mathcal{L}'$. In other words, F is induced by the strictly indexed functor:

$$(W : \mathbb{1} \rightarrow \mathcal{D}, w : \mathbb{1} \rightarrow \mathcal{L}'(W))$$

in which W and w denote the obvious functors. Since any functor $\mathbb{1} \rightarrow \mathcal{D}$ is a morphism of $\mu\nu\text{Poly}_{\mathcal{C}}$ and, for any $W \in \mathcal{D}$, any functor $\mathbb{1} \rightarrow \mathcal{L}'(W)$ is a morphism of $\mu\nu\text{Poly}_{\mathcal{L}}$, we have that $(W : \mathbb{1} \rightarrow \mathcal{D}, w : \mathbb{1} \rightarrow \mathcal{L}'(W))$ is a $\mu\nu\mathcal{L}$ -indexed functor.

- (C) By Proposition 17, the binary product $\times : \Sigma_{\mathcal{C}}\mathcal{L} \times \Sigma_{\mathcal{C}}\mathcal{L} \rightarrow \Sigma_{\mathcal{C}}\mathcal{L}$ is induced by the strictly indexed functor:

$$(\times : \mathcal{C} \times \mathcal{C} \rightarrow \mathcal{C}, p) : \mathcal{L} \underline{\times} \mathcal{L} \rightarrow \mathcal{L}$$

in which $p_{(W,W')}$ is given by the composition:

$$\begin{array}{ccc} \mathcal{L}(W) \times \mathcal{L}(W') & \xrightarrow{\mathcal{L}(\pi_1) \times \mathcal{L}(\pi_2)} & \mathcal{L}(W \times W') \times \mathcal{L}(W \times W') \\ & \searrow p_{(W,W')} & \downarrow + \\ & & \mathcal{L}(W \times W') \end{array}$$

It remains to show that $(\times : \mathcal{C} \times \mathcal{C} \rightarrow \mathcal{C}, p)$ is a $\mu\nu\mathcal{L}$ -indexed functor. Since $\times : \mathcal{C} \times \mathcal{C} \rightarrow \mathcal{C}$ is a morphism of $\mu\nu\text{Poly}_{\mathcal{C}}$, it is enough to prove that $p_{(W,W')}$ is a morphism of $\mu\nu\text{Poly}_{\mathcal{L}}$ for any $(W, W') \in \mathcal{C} \times \mathcal{C}$.

Since, for any $(W, W') \in \mathcal{C} \times \mathcal{C}$, we have that

$$\begin{aligned} \pi_{\mathcal{L}(W)} : \mathcal{L}(W) \times \mathcal{L}(W') &\rightarrow \mathcal{L}(W), & \pi_{\mathcal{L}(W')} : \mathcal{L}(W) \times \mathcal{L}(W') &\rightarrow \mathcal{L}(W') \\ \mathcal{L}(\pi_1) : \mathcal{L}(W) &\rightarrow \mathcal{L}(W \times W'), & \mathcal{L}(\pi_2) : \mathcal{L}(W') &\rightarrow \mathcal{L}(W \times W') \end{aligned}$$

are morphisms of $\mu\nu\text{Poly}_{\mathcal{L}}$, we conclude that

$$(\mathcal{L}(\pi_1) \circ \pi_{\mathcal{L}(W)}, \mathcal{L}(\pi_2) \circ \pi_{\mathcal{L}(W')}) = \mathcal{L}(\pi_1) \times \mathcal{L}(\pi_2)$$

is a morphism of $\mu\nu\text{Poly}_{\mathcal{L}}$. Thus, since $\times : \mathcal{L}(W \times W') \times \mathcal{L}(W \times W') \rightarrow \mathcal{L}(W \times W')$ is a morphism of $\mu\nu\text{Poly}_{\mathcal{L}}$ as well, we conclude that the composition $p_{(W,W')}$ is a morphism of $\mu\nu\text{Poly}_{\mathcal{L}}$.

- (D) By Corollary 35, the coproduct $\sqcup : \Sigma_{\mathcal{C}}\mathcal{L} \times \Sigma_{\mathcal{C}}\mathcal{L} \rightarrow \Sigma_{\mathcal{C}}\mathcal{L}$ is induced by the strictly indexed functor:

$$(\sqcup : \mathcal{C} \times \mathcal{C} \rightarrow \mathcal{C}, s) : \mathcal{L} \underline{\times} \mathcal{L} \rightarrow \mathcal{L}$$

in which $s_{(W,W')}$ is given by the functor:

$$\mathcal{S}^{(W,W')} : \mathcal{L}(W) \times \mathcal{L}(X) \rightarrow \mathcal{L}(W \sqcup X)$$

of the extensive structure (see (11)) is a morphism of $\mu\nu\text{Poly}_{\mathcal{L}}$.

We have that $(\sqcup : \mathcal{C} \times \mathcal{C} \rightarrow \mathcal{C}, s) : \mathcal{L} \underline{\times} \mathcal{L} \rightarrow \mathcal{L}$ is a $\mu\nu\mathcal{L}$ -indexed functor, since $\sqcup : \mathcal{C} \times \mathcal{C} \rightarrow \mathcal{C}$ is a morphism of $\mu\nu\text{Poly}_{\mathcal{C}}$ and $\mathcal{S}^{(W,W')}$ is a morphism of $\mu\nu\text{Poly}_{\mathcal{L}}$, for any $(W, W') \in \mathcal{C} \times \mathcal{C}$.

- (E) The projections

$$\pi_1 : \mathcal{A} \times \mathcal{A}' \rightarrow \mathcal{A}, \quad \pi_2 : \mathcal{A} \times \mathcal{A}' \rightarrow \mathcal{A}'$$

are, respectively, induced by the strictly indexed functors:

$$\begin{aligned} (\pi_1 : \mathcal{D} \times \mathcal{E} \rightarrow \mathcal{D}, (\pi_1 : \mathcal{L}(W) \times \mathcal{L}(W') \rightarrow \mathcal{L}(W))_{(W,W') \in \mathcal{D} \times \mathcal{E}}) : \mathcal{L}' \underline{\times} \mathcal{L}'' &\rightarrow \mathcal{L}' \\ (\pi_2 : \mathcal{D} \times \mathcal{E} \rightarrow \mathcal{E}, (\pi_2 : \mathcal{L}(W) \times \mathcal{L}(W') \rightarrow \mathcal{L}(W'))_{(W,W') \in \mathcal{D} \times \mathcal{E}}) : \mathcal{L}' \underline{\times} \mathcal{L}'' &\rightarrow \mathcal{L}'' \end{aligned}$$

which are $\mu\nu\mathcal{L}$ -indexed functors, since

$$\pi_1 : \mathcal{D} \times \mathcal{E} \rightarrow \mathcal{D}, \quad \pi_2 : \mathcal{D} \times \mathcal{E} \rightarrow \mathcal{E}$$

are morphisms of $\mu\nu\text{Poly}_{\mathcal{C}}$ and, for any $(W, W') \in \mathcal{D} \times \mathcal{E}$,

$$\pi_1 : \mathcal{L}(W) \times \mathcal{L}(W') \rightarrow \mathcal{L}(W), \quad \pi_2 : \mathcal{L}(W) \times \mathcal{L}(W') \rightarrow \mathcal{L}(W')$$

are morphisms of $\mu\nu\text{Poly}_{\mathcal{L}}$.

- (F) Assuming that $E : \mathcal{A} \rightarrow \mathcal{A}'$ and $J : \mathcal{A} \rightarrow \mathcal{A}''$ are functors induced by the $\mu\nu\mathcal{L}$ -indexed functors:

$$\left(\bar{E}, e : \mathcal{L}' \rightarrow \mathcal{L}'' \circ \bar{E}^{\text{op}}\right) : \mathcal{L}' \rightarrow \mathcal{L}'' \quad \text{and} \quad \left(\bar{J}, j : \mathcal{L}' \rightarrow \mathcal{L}''' \circ \bar{J}^{\text{op}}\right) : \mathcal{L}' \rightarrow \mathcal{L}''',$$

the functor $(E, J) : \mathcal{A} \rightarrow \mathcal{A}' \times \mathcal{A}''$ is induced by the strictly indexed functor:

$$\left((\bar{E}, \bar{J}), (e, j)\right) : \mathcal{L}' \rightarrow \mathcal{L}'' \times \mathcal{L}'''.$$

which is a $\mu\nu\mathcal{L}$ -indexed functor as well since:

- \bar{E}, \bar{J} are morphisms of $\mu\nu\text{Poly}_{\mathcal{C}}$ and, hence, so is (\bar{E}, \bar{J}) ;
- e_W, j_W are morphisms of $\mu\nu\text{Poly}_{\mathcal{L}}$ for any $W \in \mathcal{D}$ and, hence, so is (e_W, j_W) .

Finally, assuming that $H : \mathcal{A} \times \Sigma_{\mathcal{C}}\mathcal{L} \rightarrow \Sigma_{\mathcal{C}}\mathcal{L}$ is a functor induced by a $\mu\nu\mathcal{L}$ -functor:

$$\left(\bar{H}, h\right) : \mathcal{L}' \times \mathcal{L} \rightarrow \mathcal{L},$$

we have, by Theorem 57, that

- (G) μH is induced by the $\mu\nu\mathcal{L}$ -indexed functor:

$$\left(\mu\bar{H} : \mathcal{E} \rightarrow \mathcal{D}, \mu(h_{(-)})\right) : \mathcal{L}' \rightarrow \mathcal{L}.$$

- (H) νH is induced by the $\mu\nu\mathcal{L}$ -indexed functor:

$$\left(\nu\bar{H} : \mathcal{E} \rightarrow \mathcal{D}, \nu(\bar{h}_{(-)})\right) : \mathcal{L}' \rightarrow \mathcal{L}. \quad \square$$

Codually, we have:

Theorem 64. *Let $\mathcal{L} : \mathcal{C}^{\text{op}} \rightarrow \mathbf{Cat}$ be a $*$ -indexed category. The category $\Sigma_{\mathcal{C}}\mathcal{L}^{\text{op}}$ has $\mu\nu$ -polynomials.*

6.13 Σ -bimodel for function types, inductive and coinductive types

By Theorem 39, the Grothendieck construction of any Σ -bimodel for inductive and coinductive types is distributive. Moreover, we get the closed structure if \mathcal{L} satisfies the conditions of Section 6.4. More precisely:

Corollary 65. *Let $\mathcal{L} : \mathcal{C}^{\text{op}} \rightarrow \mathbf{Cat}$ be a Σ -bimodel for inductive and coinductive types. The categories $\Sigma_{\mathcal{C}}\mathcal{L}$ and $\Sigma_{\mathcal{C}}\mathcal{L}^{\text{op}}$ are distributive categories with $\mu\nu$ -polynomials.*

Corollary 66. *Let $\mathcal{L} : \mathcal{C}^{\text{op}} \rightarrow \mathbf{Cat}$ be a Σ -bimodel for inductive, coinductive, and function types. The categories $\Sigma_{\mathcal{C}}\mathcal{L}$ and $\Sigma_{\mathcal{C}}\mathcal{L}^{\text{op}}$ are closed categories with $\mu\nu$ -polynomials.*

7. Linear λ -Calculus as an Idealized AD Target Language

We describe a target language for our AD code transformations, a variant of the dependently typed enriched effect calculus (Vákár 2017, Chapter 5). Its cartesian types, linear types, and terms

$\kappa, \kappa', \kappa'' ::=$ type ltype	kinds kind of Cartesian types kind of linear types
$\underline{\tau}, \underline{\sigma}, \underline{\rho} ::=$ $\underline{\alpha}$ real ⁿ $\underline{1}$ $\underline{\tau} * \underline{\sigma}$ $\Pi x : \tau. \underline{\sigma}$ $\Sigma x : \tau. \underline{\sigma}$ case t of { $\ell_1 x_1 \rightarrow \underline{\tau}_1 \mid \dots \mid \ell_n x_n \rightarrow \underline{\tau}_n$ } $\underline{\mu \alpha}. \underline{\tau}$ $\underline{\nu \alpha}. \underline{\tau}$	linear types linear type variable real array unit type binary product power copower case distinction inductive type coinductive type
$\tau, \sigma, \rho ::=$... $\underline{\tau} \multimap \underline{\sigma}$ $\Pi x : \tau. \sigma$ $\Sigma x : \tau. \sigma$	Cartesian types as in Fig. 1 linear function dependent function dependent pair
$t, s, r ::=$ terms ... v let $v = t$ in s $\text{lop}(t_1, \dots, t_k; s)$ $!t \otimes s \mid \text{case } t \text{ of } !y \otimes v \rightarrow s$ $\underline{\lambda} v. t \mid t \bullet s$ $\underline{0} \mid t + s$	as in Fig. 1 linear identifier linear let-binding linear operation copower intro/elim abstraction/application monoid structure

Figure 6. A grammar for the kinds, types, and terms of the target language, extending that of Fig. 1.

are generated by the grammar of Figs. 1 and 6, making the target language a proper extension of the source language. We note that we use a special symbol v for the unique linear identifier. We introduce kinding judgments $\Delta \mid \Gamma \vdash \tau : \text{type}$ and $\Delta \mid \Gamma \vdash \underline{\alpha} : \text{ltype}$ for cartesian and linear types, where $\Delta = \alpha_1 : \text{type}, \dots, \alpha_n : \text{type}$ is a list of (cartesian) type identifiers and $\Gamma = x_1 : \tau_1, \dots, x_n : \tau_n$ is a list of identifiers x_i with cartesian type τ_i . These kinding judgments are defined according to the rules displayed in Figs. 2 and 7.

We use typing judgments $\Delta \mid \Gamma \vdash t : \tau$ and $\Delta \mid \Gamma; v : \underline{\alpha} \vdash s : \underline{\sigma}$ for terms of well-kinded cartesian types $\Delta \mid \Gamma \vdash \tau : \text{type}$ and linear type $\Delta \mid \Gamma \vdash \underline{\alpha} : \text{ltype}$, where $\Delta = \alpha_1 : \text{type}, \dots, \alpha_n : \text{type}$ is a list of cartesian type identifiers, $\Gamma = x_1 : \tau_1, \dots, x_n : \tau_n$ is a list of identifiers x_i of well-kinded cartesian type $\Delta \mid x_1 : \tau_1, \dots, x_{i-1} : \tau_{i-1} \vdash \tau_i : \text{type}$ and v is the unique linear identifier of well-kinded linear type $\Delta \mid \Gamma \vdash \underline{\alpha} : \text{ltype}$. Note that terms of linear type always contain the unique linear identifier v in the typing context. These typing judgments are defined according to the rules displayed in Figs. 3, 8 and 9.

We work with linear operations $\text{lop} \in \text{LOp}_{n_1, \dots, n_k; n'_1, \dots, n'_l}^{m_1, \dots, m_r}$, which are intended to represent functions that are linear (in the sense of respecting $\underline{0}$ and $+$) in the last l arguments but not in the

$$\begin{array}{c}
 \frac{\Delta \vdash \tau : \text{type}}{\Delta \mid \Gamma \vdash \tau : \text{type}} \quad \frac{((\alpha : \text{type}) \in \Delta)}{\Delta \mid \Gamma \vdash \underline{\alpha} : \text{ltype}} \\
 \\
 \frac{}{\Delta \mid \Gamma \vdash \mathbf{real}^n : \text{ltype}} \quad \frac{}{\Delta \mid \Gamma \vdash \mathbf{1} : \text{ltype}} \quad \frac{\Delta \mid \Gamma \vdash \underline{\tau} : \text{ltype} \quad \Delta \mid \Gamma \vdash \underline{\sigma} : \text{ltype}}{\Delta \mid \Gamma \vdash \underline{\tau} * \underline{\sigma} : \text{ltype}} \\
 \\
 \frac{\cdot \mid \Gamma \vdash \tau : \text{type} \quad \cdot \mid \Gamma, x : \tau \vdash \underline{\sigma} : \text{ltype}}{\cdot \mid \Gamma \vdash \Pi x : \tau. \underline{\sigma} : \text{ltype}} \quad \frac{\cdot \mid \Gamma \vdash \tau : \text{type} \quad \cdot \mid \Gamma, x : \tau \vdash \underline{\sigma} : \text{ltype}}{\cdot \mid \Gamma \vdash \Sigma x : \tau. \underline{\sigma} : \text{ltype}} \\
 \\
 \frac{\Delta \mid \Gamma \vdash t : \{\ell_1 \tau_1 \mid \dots \mid \ell_n \tau_n\} \quad \{\Delta \mid \Gamma, x_i : \tau_i \vdash \underline{\sigma}_i : \text{ltype}\}_{1 \leq i \leq n}}{\Delta \mid \Gamma \vdash \mathbf{case } t \mathbf{ of } \{\ell_1 x_1 \rightarrow \underline{\sigma}_1 \mid \dots \mid \ell_n x_n \rightarrow \underline{\sigma}_n\} : \text{ltype}} \\
 \\
 \frac{\Delta, \alpha : \text{type} \mid \Gamma \vdash \underline{\tau} : \text{ltype}}{\Delta \mid \Gamma \vdash \underline{\mu \alpha. \tau} : \text{ltype}} \quad \frac{\Delta, \alpha : \text{type} \mid \Gamma \vdash \underline{\tau} : \text{ltype}}{\Delta \mid \Gamma \vdash \underline{\nu \alpha. \tau} : \text{ltype}} \\
 \\
 \frac{\cdot \mid \Gamma \vdash \underline{\tau} : \text{ltype} \quad \cdot \mid \Gamma \vdash \underline{\sigma} : \text{ltype}}{\cdot \mid \Gamma \vdash \underline{\tau} \multimap \underline{\sigma} : \text{type}} \\
 \\
 \frac{\cdot \mid \Gamma \vdash \tau : \text{type} \quad \cdot \mid \Gamma, x : \tau \vdash \sigma : \text{type}}{\cdot \mid \Gamma \vdash \Pi x : \tau. \sigma : \text{type}} \quad \frac{\cdot \mid \Gamma \vdash \tau : \text{type} \quad \cdot \mid \Gamma, x : \tau \vdash \sigma : \text{type}}{\cdot \mid \Gamma \vdash \Sigma x : \tau. \sigma : \text{type}}
 \end{array}$$

Figure 7. Kinding rules for the AD target language that we consider on top of those of Fig. 2, where our first rule specifies how kinding judgments of the source language imply kinding of types in the target language. Observe that, according to the second rule, type variables α from the kinding context Δ can be used as a linear type $\underline{\alpha}$. Note that we only consider the formation of Σ - and Π -types and linear function types of nonparameterized types (shaded in gray).

first k . To serve as a practical target language for the automatic derivatives of all programs from the source language, we work with the following linear operations: for all $\text{op} \in \text{Op}_{n_1, \dots, n_k}^m$:

$$\text{Dop} \in \text{LOp}_{n_1, \dots, n_k; n_1, \dots, n_k}^m \quad \text{Dop}^t = (\text{Dop})^t \in \text{LOp}_{n_1, \dots, n_k; m}^{n_1, \dots, n_k}$$

We will use these linear operations Dop and Dop^t as the forward and reverse derivatives of the corresponding primitive operations op ⁶. We write

$$\mathbf{LDom}(\text{lop}) \stackrel{\text{def}}{=} \mathbf{real}^{n'_1} * \dots * \mathbf{real}^{n'_l} \quad \text{and} \quad \mathbf{CDom}(\text{lop}) \stackrel{\text{def}}{=} \mathbf{real}^{m_1} * \dots * \mathbf{real}^{m_r}$$

for $\text{lop} \in \text{LOp}_{n_1, \dots, n_k; n'_1, \dots, n'_l}^{m_1, \dots, m_r}$.

Figs. 4 and 10 display the equational theory we consider for the terms and types, which we call $(\alpha)\beta\eta$ -equivalence. To present this equational theory, we define in Fig. 11, by induction, some syntactic sugar for the functorial action $\Delta, \Delta' \mid \Gamma; \nu : \underline{\alpha}[\frac{\sigma}{\alpha}] \vdash \underline{\alpha}[\frac{\nu \vdash t}{\alpha}] : \underline{\alpha}[\frac{\gamma}{\alpha}]$ in argument $\underline{\alpha}$ of parameterized types $\Delta, \alpha : \text{type} \mid \Gamma \vdash \underline{\alpha} : \text{ltype}$ on terms $\Delta' \mid \Gamma; \nu : \underline{\sigma} \vdash t : \underline{\gamma}$.

This target language can be viewed as defining a strictly indexed category $\mathbf{LSyn} : \mathbf{CSyn}^{\text{op}} \rightarrow \mathbf{Cat}$:

- \mathbf{CSyn} extends its full subcategory \mathbf{Syn} with the newly added cartesian types; its objects are cartesian types, and $\mathbf{CSyn}(\tau, \sigma)$ consists of $(\alpha)\beta\eta$ -equivalence classes of target language programs $\cdot \mid x : \tau \vdash t : \sigma$.
- Objects of $\mathbf{LSyn}(\tau)$ are linear types $\cdot \mid p : \tau \vdash \underline{\sigma} : \text{ltype}$ up to $(\alpha)\beta\eta$ -equivalence.
- Morphisms in $\mathbf{LSyn}(\tau)(\underline{\sigma}, \underline{\gamma})$ are terms $\cdot \mid x : \tau; \nu : \underline{\sigma} \vdash t : \underline{\gamma}$ modulo $(\alpha)\beta\eta$ -equivalence.
- Identities in $\mathbf{LSyn}(\tau)$ are represented by the terms $\cdot \mid x : \tau; \nu : \underline{\sigma} \vdash \nu : \underline{\sigma}$.

$$\begin{array}{c}
 \frac{}{\Delta | \Gamma; \underline{\tau} \vdash \underline{\tau} : \underline{\tau}} \quad \frac{\Delta | \Gamma; \underline{v}; \underline{\tau} \vdash t : \underline{\sigma} \quad \Delta | \Gamma; \underline{v}; \underline{\sigma} \vdash s : \underline{\rho}}{\Delta | \Gamma; \underline{v}; \underline{\tau} \vdash \mathbf{let} \, v = t \, \mathbf{in} \, s : \underline{\rho}} \quad \frac{\Delta | \Gamma \vdash t : \underline{\tau} \quad \Delta | \Gamma, x : \underline{\tau}; \underline{v}; \underline{\sigma} \vdash s : \underline{\rho}}{\Delta | \Gamma; \underline{v}; \underline{\sigma} [\underline{t}/x] \vdash \mathbf{let} \, x = t \, \mathbf{in} \, s : \underline{\rho} [\underline{t}/x]} \\
 \\
 \frac{\{ \Delta | \Gamma \vdash t_i : \mathbf{real}^{n_i} \}_{i=1}^k \quad \Delta | \Gamma; \underline{v}; \underline{\tau} \vdash s : \mathbf{LDom}(\text{lop}) \quad (\text{lop} \in \text{LOP}_{n_1, \dots, n_k; n'_1, \dots, n'_l}^{m_1, \dots, m_r})}{\Delta | \Gamma; \underline{v}; \underline{\tau} \vdash \text{lop}(t_1, \dots, t_k; s) : \mathbf{CDom}(\text{lop})} \\
 \\
 \frac{}{\Delta | \Gamma; \underline{v}; \underline{\tau} \vdash \langle \rangle : \underline{\mathbf{1}}} \quad \frac{\Delta | \Gamma; \underline{v}; \underline{\tau} \vdash t : \underline{\sigma} \quad \Delta | \Gamma; \underline{v}; \underline{\tau} \vdash s : \underline{\rho}}{\Delta | \Gamma; \underline{v}; \underline{\tau} \vdash (t, s) : \underline{\sigma} * \underline{\rho}} \quad \frac{\Delta | \Gamma; \underline{v}; \underline{\tau} \vdash t : \underline{\sigma} * \underline{\rho}}{\Delta | \Gamma; \underline{v}; \underline{\tau} \vdash \mathbf{fst} \, t : \underline{\sigma}} \\
 \\
 \frac{\Delta | \Gamma; \underline{v}; \underline{\tau} \vdash t : \underline{\sigma} * \underline{\rho}}{\Delta | \Gamma; \underline{v}; \underline{\tau} \vdash \mathbf{snd} \, t : \underline{\rho}} \quad \frac{\Delta | \Gamma, y : \underline{\sigma}; \underline{v}; \underline{\tau} \vdash t : \underline{\rho}}{\Delta | \Gamma; \underline{v}; \underline{\tau} \vdash \lambda y. t : \Pi y : \underline{\sigma}. \underline{\rho}} \\
 \\
 \frac{\Delta | \Gamma; \underline{v}; \underline{\tau} \vdash t : \Pi y : \underline{\sigma}. \underline{\rho} \quad \Delta | \Gamma \vdash s : \underline{\sigma}}{\Delta | \Gamma; \underline{v}; \underline{\tau} \vdash t \, s : \underline{\rho} [\underline{s}/y]} \quad \frac{\Delta | \Gamma \vdash t : \underline{\sigma} \quad \Delta | \Gamma; \underline{v}; \underline{\tau} \vdash s : \underline{\rho} [\underline{t}/y]}{\Delta | \Gamma; \underline{v}; \underline{\tau} \vdash t \otimes s : \Sigma y : \underline{\sigma}. \underline{\rho}} \\
 \\
 \frac{\Delta | \Gamma; \underline{v}; \underline{\tau} \vdash t : \Sigma y : \underline{\sigma}. \underline{\rho} \quad \Delta | \Gamma, y : \underline{\sigma}; \underline{v}; \underline{\rho} \vdash s : \underline{\rho}' \quad \Delta | \Gamma | \cdot \vdash \underline{\rho}' : \text{ltype}}{\Delta | \Gamma; \underline{v}; \underline{\tau} \vdash \mathbf{case} \, t \, \mathbf{of} \, !y \otimes v \rightarrow s : \underline{\rho}'} \quad \frac{\Delta | \Gamma; \underline{v}; \underline{\tau} \vdash t : \underline{\sigma}}{\Delta | \Gamma \vdash \underline{\lambda} v. t : \underline{\tau} \rightarrow \underline{\sigma}} \\
 \\
 \frac{\Delta | \Gamma \vdash t : \underline{\rho} \rightarrow \underline{\sigma} \quad \Delta | \Gamma; \underline{v}; \underline{\tau} \vdash s : \underline{\rho}}{\Delta | \Gamma; \underline{v}; \underline{\tau} \vdash t \bullet s : \underline{\sigma}} \quad \frac{}{\Delta | \Gamma; \underline{v}; \underline{\tau} \vdash \underline{0} : \underline{\sigma}} \quad \frac{\Delta | \Gamma; \underline{v}; \underline{\tau} \vdash t : \underline{\sigma} \quad \Delta | \Gamma; \underline{v}; \underline{\tau} \vdash s : \underline{\sigma}}{\Delta | \Gamma; \underline{v}; \underline{\tau} \vdash t + s : \underline{\sigma}} \\
 \\
 \frac{\Delta | x : \{ \ell_1 \tau_1 | \dots | \ell_n \tau_n \} \vdash \underline{\sigma}, \underline{\rho} : \text{ltype} \quad \Delta | \Gamma \vdash t : \{ \ell_1 \tau_1 | \dots | \ell_n \tau_n \} \quad \left\{ \Delta | x_i : \tau_i; \underline{v}; \underline{\sigma} [\underline{\ell}_i x_i / x] \vdash r_i : \underline{\rho} [\underline{\ell}_i x_i / x] \right\}_{1 \leq i \leq n}}{\Delta | \Gamma; \underline{\sigma} [\underline{t}/x] \vdash \mathbf{case} \, t \, \mathbf{of} \, \{ \ell_1 x_1 \rightarrow r_1 | \dots | \ell_n x_n \rightarrow r_n \} : \underline{\rho} [\underline{t}/x]} \\
 \\
 \frac{\Delta | \Gamma; \underline{v}; \underline{\rho} \vdash t : \underline{\tau} [\underline{\mu} \underline{\alpha}. \underline{\tau} / \underline{\alpha}]}{\Delta | \Gamma; \underline{v}; \underline{\rho} \vdash \mathbf{roll} \, t : \underline{\mu} \underline{\alpha}. \underline{\tau}} \quad \frac{\Delta | \Gamma; \underline{v}; \underline{\rho} \vdash t : \underline{\mu} \underline{\alpha}. \underline{\tau} \quad \Delta | \Gamma; \underline{v}; \underline{\tau} [\underline{\sigma} / \underline{\alpha}] \vdash s : \underline{\sigma}}{\Delta | \Gamma; \underline{v}; \underline{\rho} \vdash \mathbf{fold} \, t \, \mathbf{with} \, v \rightarrow s : \underline{\sigma}} \\
 \\
 \frac{\Delta | \Gamma; \underline{v}; \underline{\rho} \vdash t : \underline{\sigma} \quad \Delta | \Gamma; \underline{v}; \underline{\sigma} \vdash s : \underline{\tau} [\underline{\sigma} / \underline{\alpha}]}{\Delta | \Gamma; \underline{v}; \underline{\rho} \vdash \mathbf{gen} \, \mathbf{from} \, t \, \mathbf{with} \, v \rightarrow s : \underline{\nu} \underline{\alpha}. \underline{\tau}} \quad \frac{\Delta | \Gamma; \underline{v}; \underline{\rho} \vdash t : \underline{\nu} \underline{\alpha}. \underline{\tau}}{\Delta | \Gamma; \underline{v}; \underline{\rho} \vdash \mathbf{unroll} \, t : \underline{\tau} [\underline{\nu} \underline{\alpha}. \underline{\tau} / \underline{\alpha}]}
 \end{array}$$

Figure 8. Typing rules for the AD target language that we consider on top of the rules of Figs. 3 and 9.

- Composition of $\cdot | x : \tau; \underline{v}; \underline{\sigma}_1 \vdash t : \underline{\sigma}_2$ and $\cdot | x : \tau; \underline{v}; \underline{\sigma}_2 \vdash s : \underline{\sigma}_3$ in $\mathbf{LSyn}(\tau)$ is defined as $\cdot | x : \tau; \underline{v}; \underline{\sigma}_1 \vdash \mathbf{let} \, v = t \, \mathbf{in} \, s : \underline{\sigma}_3$.
- Change of base $\mathbf{LSyn}(t) : \mathbf{LSyn}(\tau) \rightarrow \mathbf{LSyn}(\tau')$ along $(\cdot | x' : \tau' \vdash t : \tau) \in \mathbf{CSyn}(\tau', \tau)$ is defined $\mathbf{LSyn}(t) \cdot (\cdot | x : \tau; \underline{v}; \underline{\sigma} \vdash s : \underline{\gamma}) \stackrel{\text{def}}{=} \cdot | x' : \tau'; \underline{v}; \underline{\sigma} \vdash \mathbf{let} \, x = t \, \mathbf{in} \, s : \underline{\gamma}$.
- All type formers are interpreted as one expects based on their notation, using introduction and elimination rules for the required structural isomorphisms.

Corollary 67. $\Sigma_{\mathbf{CSyn} \mathbf{LSyn}}$ and $\Sigma_{\mathbf{CSyn} \mathbf{LSyn}^{op}}$ are both bicartesian closed categories with $\mu\nu$ -polynomials.

In fact, $\mathbf{LSyn} : \mathbf{CSyn}^{op} \rightarrow \mathbf{Cat}$ is the initial Σ -bimodel of tuples, self-dual primitive types and primitive operations, function types, sum types, and inductive and coinductive types, in the sense

$$\frac{\Delta \mid \Gamma \vdash t : \tau \quad \Delta \mid \Gamma \vdash s : \rho}{\Delta \mid \Gamma \vdash \mathbf{let} \ x = t \ \mathbf{in} \ s : \rho[t/x]}$$

$$\frac{\Delta \mid \Gamma, x : \tau \vdash t : \sigma}{\Delta \mid \Gamma \vdash \lambda x. t : \Pi x : \tau. \sigma} \quad \frac{\Delta \mid \Gamma \vdash t : \Pi x : \tau. \sigma \quad \Delta \mid \Gamma \vdash s : \tau}{\Delta \mid \Gamma \vdash t \ s : \sigma[t/x]}$$

$$\frac{\Delta \mid \Gamma \vdash t : \tau \quad \Delta \mid \Gamma \vdash s : \sigma[t/x]}{\Delta \mid \Gamma \vdash \langle t, s \rangle : \Sigma x : \tau. \sigma} \quad \frac{\Delta \mid \Gamma \vdash t : \Sigma x : \tau. \sigma}{\Delta \mid \Gamma \vdash \mathbf{fst} \ t : \tau} \quad \frac{\Delta \mid \Gamma \vdash t : \Sigma x : \tau. \sigma}{\Delta \mid \Gamma \vdash \mathbf{snd} \ t : \sigma[\mathbf{fst} \ t/x]}$$

Figure 9. Typing rules for the AD target language that we consider on top of the rules of Figs. 3 and 8.

$$\mathbf{let} \ v = t \ \mathbf{in} \ s = s[t/v]$$

$$\mathbf{case} \ !t \otimes s \ \mathbf{of} \ !x \otimes v \rightarrow r = r[t/x, s/v] \quad t[s/v] \stackrel{\#y}{=} \mathbf{case} \ s \ \mathbf{of} \ !y \otimes v \rightarrow t[!y \otimes v/v]$$

$$\langle \underline{\lambda} v. t \rangle \bullet s = t[s/v] \quad t = \underline{\lambda} v. t \bullet v$$

$$t + \underline{0} = t \quad \underline{0} + t = t \quad (t + s) + r = t + (s + r) \quad t + s = s + t$$

$$(\Gamma; v : \underline{\tau} \vdash t : \underline{\sigma}) \ \text{implies} \ t[\underline{0}/v] = \underline{0} \quad (\Gamma; v : \underline{\tau} \vdash t : \underline{\sigma}) \ \text{implies} \ t[s+r/v] = t[s/v] + t[r/v]$$

$$\mathbf{case} \ \ell_i t \ \mathbf{of} \ \{\ell_1 x_1 \rightarrow \underline{\tau}_1 \mid \dots \mid \ell_n x_n \rightarrow \underline{\tau}_n\} = \underline{\tau}_i[t/x_i]$$

$$\underline{\tau}[t/y] \stackrel{\#x_1, \dots, x_n}{=} \mathbf{case} \ t \ \mathbf{of} \ \{\ell_1 x_1 \rightarrow \underline{\tau}[\ell_1 x_1/y] \mid \dots \mid \ell_n x_n \rightarrow \underline{\tau}[\ell_n x_n/y]\}$$

$$\mathbf{fold} \ \mathbf{roll} \ t \ \mathbf{with} \ v \rightarrow s = s[\underline{\tau}[\mathbf{fold} \ v \ \mathbf{with} \ v \rightarrow s/\underline{\alpha}][t/v]/v]$$

$$r[\mathbf{roll} \ v/v] = s[\underline{\tau}[\mathbf{roll} \ v/\underline{\alpha}]/v] \ \text{implies} \ r[t/v] = \mathbf{fold} \ t \ \mathbf{with} \ v \rightarrow s$$

$$\mathbf{unroll} \ (\mathbf{gen} \ \mathbf{from} \ t \ \mathbf{with} \ v \rightarrow s) = \underline{\tau}[\mathbf{v-gen} \ \mathbf{from} \ v \ \mathbf{with} \ v \rightarrow s/\underline{\alpha}][s/v, t/v]$$

$$\mathbf{unroll} \ r = \underline{\tau}[\mathbf{v-roll} \ v/\underline{\alpha}][s/v] \ \text{implies} \ r[t/v] = \mathbf{gen} \ \mathbf{from} \ t \ \mathbf{with} \ v \rightarrow s$$

Figure 10. Equational rules for the idealized, linear AD language, which we use on top of the rules of Fig. 4. In addition to standard $\beta\eta$ -rules for $!(-)\otimes_{(-)}$ - and \multimap -types, we add rules making $(0, +)$ into a commutative monoid on the terms of each linear type as well as rules which say that terms of linear types are homomorphisms in their linear variable. Equations hold on pairs of terms of the same type/types of the same kind. As usual, we only distinguish terms up to α -renaming of bound variables.

that for any other such a Σ -bimodel $\mathcal{L} : \mathcal{C}^{op} \rightarrow \mathbf{Cat}$, we have a unique homomorphic indexed functor $(\bar{F}, f) : (\mathbf{CSyn}, \mathbf{LSyn}) \rightarrow (\mathcal{C}, \mathcal{L})$.

Corollary 68 (Concrete semantics of the target language). *Let $\mathcal{L} : \mathcal{C}^{op} \rightarrow \mathbf{Cat}$ be a Σ -bimodel for inductive, coinductive and function types. Let*

- (a) for each n -dimensional array $\mathbf{real}^n \in \mathbf{Syn}$, $\bar{F}(\mathbf{real}^n) \in \text{obj}(\mathcal{C})$;
- (b) for each n -dimensional array $\mathbf{real}^n \in \mathbf{Syn}$,

$$\underline{F}(\mathbf{real}^n) \in \mathcal{L}(\bar{F}(\mathbf{real}^n));$$
- (c) for each primitive $\text{op} \in \text{Op}_{n_1, \dots, n_k}^m$:

$$\begin{aligned}
 \underline{\alpha}[\underline{v}^t/\underline{\alpha}] &= t \\
 \underline{\beta}[\underline{v}^t/\underline{\alpha}] &= v \quad \text{if } \underline{\alpha} \neq \underline{\beta} \\
 \mathbf{real}^n[\underline{v}^t/\underline{\alpha}] &= v \\
 \underline{1}[\underline{v}^t/\underline{\alpha}] &= v \\
 (\underline{\tau} * \underline{\sigma})[\underline{v}^t/\underline{\alpha}] &= \langle \underline{\tau}[\underline{v}^t/\underline{\alpha}][\mathbf{fst} v/v], \underline{\sigma}[\underline{v}^t/\underline{\alpha}][\mathbf{snd} v/v] \rangle \\
 \left(\mathbf{case\ s\ of} \begin{array}{l} \ell_1 x_1 \rightarrow \underline{\tau}_1 \\ \vdots \\ \ell_n x_n \rightarrow \underline{\tau}_n \end{array} \right) [\underline{v}^t/\underline{\alpha}] &= \mathbf{case\ s\ of} \begin{array}{l} \ell_1 x_1 \rightarrow \underline{\tau}_1[\underline{v}^t/\underline{\alpha}] \\ \vdots \\ \ell_n x_n \rightarrow \underline{\tau}_n[\underline{v}^t/\underline{\alpha}] \end{array} \\
 (\underline{\mu}\underline{\alpha}.\underline{\tau})[\underline{v}^t/\underline{\alpha}] &= v \\
 (\underline{\mu}\underline{\beta}.\underline{\tau})[\underline{v}^t/\underline{\alpha}] &= \mathbf{fold\ v\ with\ } v \rightarrow \mathbf{roll\ } \underline{\tau}[\underline{v}^t/\underline{\alpha}] \quad \text{if } \underline{\alpha} \neq \underline{\beta} \\
 (\underline{\nu}\underline{\alpha}.\underline{\tau})[\underline{v}^t/\underline{\alpha}] &= v \\
 (\underline{\nu}\underline{\beta}.\underline{\tau})[\underline{v}^t/\underline{\alpha}] &= \mathbf{gen\ from\ } v \mathbf{ with\ } v \rightarrow \underline{\tau}[\underline{v}^t/\underline{\alpha}][\mathbf{unroll} v/v] \quad \text{if } \underline{\alpha} \neq \underline{\beta}
 \end{aligned}$$

Figure 11. Functorial action $\Delta, \Delta' \mid \Gamma; v : \underline{\alpha}[\underline{\sigma}/\underline{\alpha}] \vdash \underline{\alpha}[\underline{v}^t/\underline{\alpha}] : \underline{\alpha}[\underline{\nu}/\underline{\alpha}]$ in argument $\underline{\alpha}$ of parameterized types $\Delta, \alpha : \text{type} \mid \Gamma \vdash \underline{\alpha} : \text{ltype}$ on terms $\Delta' \mid \Gamma; v : \underline{\alpha} \vdash t : \underline{\nu}$ of the target language.

- (i) $\bar{F}(\text{op}) : \mathbb{R}^{n_1} \times \dots \times \mathbb{R}^{n_k} \rightarrow \mathbb{R}^m$ is the map in **Set** corresponding to the operation that *op* intends to implement;
- (ii) $f_{\text{op}} \in \mathbf{FVect}(\bar{F}(\mathbf{real}^{n_1}) \times \dots \times \bar{F}(\mathbf{real}^{n_k}), (\underline{F}(\mathbf{real}^{n_1}) \times \dots \times \underline{F}(\mathbf{real}^{n_k}), \underline{F}(\mathbf{real}^m)))$ is the family of linear transformations that *Dop* intends to implement;
- (iii) $f_{\text{op}}^t \in \mathbf{FVect}(\bar{F}(\mathbf{real}^{n_1}) \times \dots \times \bar{F}(\mathbf{real}^{n_k}), (\underline{F}(\mathbf{real}^m), \underline{F}(\mathbf{real}^{n_1}) \times \dots \times \underline{F}(\mathbf{real}^{n_k})))$ is the family of linear transformations that $(\text{Dop})^t$ intends to implement.

be an assignment. We obtain canonical bicartesian closed functors that preserve $\mu\nu$ -polynomials:

$$F : \Sigma_{\mathbf{CSyn}} \mathbf{LSyn} \rightarrow \Sigma_{\mathcal{C}} \mathcal{L} \quad (54) \qquad \qquad \qquad {}^t F : \Sigma_{\mathcal{C}} \mathbf{LSyn}^{op} \rightarrow \Sigma_{\mathbf{CSyn}} \mathcal{L}^{op} \quad (55)$$

that extend the assignment given by (a), (b), and (c).

8. Novel AD Algorithms as Source Code Transformations

By Corollary 67, $\Sigma_{\mathbf{CSyn}} \mathbf{LSyn}$ and $\Sigma_{\mathbf{CSyn}} \mathbf{LSyn}^{op}$ are both bicartesian closed categories with $\mu\nu$ -polynomials. By the universal property of **Syn** established in Corollary 15, we get unique $\mu\nu$ -polynomial-preserving bicartesian closed functors $\vec{D}(-) : \mathbf{Syn} \rightarrow \Sigma_{\mathbf{CSyn}} \mathbf{LSyn}$ and $\overleftarrow{D}(-) : \mathbf{Syn} \rightarrow \Sigma_{\mathbf{CSyn}} \mathbf{LSyn}^{op}$ implementing source code transformations for forward and reverse AD, respectively, once we fix a compatible definition for the code transformations on primitive types \mathbf{real}^n and operations *op*.

Corollary 69 (CHAD). *Once we fix the derivatives of the ground types and primitive operations of **Syn** by defining*

- for each *n*-dimensional array $\mathbf{real}^n \in \mathbf{Syn}$, $\vec{D}(\mathbf{real}^n) \stackrel{\text{def}}{=} (\mathbf{real}^n, \underline{\mathbf{real}}^n)$ and $\overleftarrow{D}(\mathbf{real}^n) \stackrel{\text{def}}{=} (\mathbf{real}^n, \underline{\mathbf{real}}^n)$ in which we think of $\underline{\mathbf{real}}^n$ as the associated tangent and cotangent space;
- for each primitive *op* $\in \text{Op}_{n_1, \dots, n_k}^m$, $\vec{D}(\text{op}) \stackrel{\text{def}}{=} (\text{op}, \text{Dop})$ and $\overleftarrow{D}(\text{op}) \stackrel{\text{def}}{=} (\text{op}, \text{Dop}^t)$, in which *Dop* and *Dop*^t are the linear operations that implement the derivative and the transposed derivative of *op*, respectively,

we obtain unique functors:

$$\vec{D}(-) : \mathbf{Syn} \rightarrow \Sigma_{\mathbf{CSyn}} \mathbf{LSyn}, \qquad \overleftarrow{D}(-) : \mathbf{Syn} \rightarrow \Sigma_{\mathbf{CSyn}} \mathbf{LSyn}^{op} \quad (56)$$

that extend these definitions such that $\overrightarrow{\mathcal{D}}(-)$ and $\overleftarrow{\mathcal{D}}(-)$ strictly preserve the bicartesian closed structure and the $\mu\nu$ -polynomials.

By definition of equality in **Syn**, $\Sigma_{\text{CSyn}}\mathbf{LSyn}$ and $\Sigma_{\text{CSyn}}\mathbf{LSyn}^{op}$, these code transformations automatically respect equational reasoning principles, in the sense that $t \stackrel{\beta\eta}{=} s$ implies that $\overrightarrow{\mathcal{D}}(t) \stackrel{\beta\eta}{=} \overrightarrow{\mathcal{D}}(s)$ and $\overleftarrow{\mathcal{D}}(t) \stackrel{\beta\eta}{=} \overleftarrow{\mathcal{D}}(s)$. In this section, we detail the implied definitions of $\overrightarrow{\mathcal{D}}$ and $\overleftarrow{\mathcal{D}}$ as well as their properties.

8.1 Some notation

In the rest of this section, we use the following syntactic sugar:

- a notation for (linear) n -ary tuple types: $(\underline{\alpha}_1 * \dots * \underline{\alpha}_n) \stackrel{\text{def}}{=} (((\underline{\alpha}_1 * \underline{\alpha}_2) \dots * \underline{\alpha}_{n-1}) * \underline{\alpha}_n)$;
- a notation for n -ary tuples: $\langle t_1, \dots, t_n \rangle \stackrel{\text{def}}{=} \langle \langle t_1, t_2 \rangle \dots, t_{n-1} \rangle, t_n$;
- given $\Gamma; \nu : \underline{\alpha} \vdash t : (\underline{\sigma}_1 * \dots * \underline{\sigma}_n)$, we write $\Gamma; \nu : \underline{\alpha} \vdash \mathbf{proj}_i(t) : \underline{\alpha}_i$ for the obvious i -th projection of t , which is constructed by repeatedly applying **fst** and **snd** to t ;
- given $\Gamma; \nu : \underline{\alpha} \vdash t : \underline{\sigma}_i$, we write the i -th coprojection $\Gamma; \nu : \underline{\alpha} \vdash \mathbf{coproj}_i(t) \stackrel{\text{def}}{=} \langle 0, \dots, 0, t, 0, \dots, 0 \rangle : (\underline{\sigma}_1 * \dots * \underline{\sigma}_n)$;
- for a list x_1, \dots, x_n of distinct identifiers, we write $\mathbf{idx}(x_i; x_1, \dots, x_n) \stackrel{\text{def}}{=} i$ for the index of the identifier x_i in this list;
- a **let**-binding for tuples: $\mathbf{let} \langle x, y \rangle = t \mathbf{in} s \stackrel{\text{def}}{=} \mathbf{let} z = t \mathbf{in} \mathbf{let} x = \mathbf{fst} z \mathbf{in} \mathbf{let} y = \mathbf{snd} z \mathbf{in} s$, where z is a fresh variable.

Furthermore, all variables used in the source code transforms below are assumed to be freshly chosen.

8.2 Kinding and typing of the code transformations

We define for each type τ of the source language:

- a cartesian type $\mathcal{D}(\tau)_1$ of forward-mode primals;
- a linear type $\overrightarrow{\mathcal{D}}(\tau)_2$ (with free term variable p) of forward-mode tangents;
- a cartesian type $\overleftarrow{\mathcal{D}}(\tau)_1$ of reverse-mode primals;
- a linear type $\overleftarrow{\mathcal{D}}(\tau)_2$ (with free term variable p) of reverse-mode cotangents.

We extend $\overrightarrow{\mathcal{D}}(-)$ and $\overleftarrow{\mathcal{D}}(-)$ to act on typing contexts $\Gamma = x_1 : \tau_1, \dots, x_n : \tau_n$ as:

$$\begin{aligned} \overrightarrow{\mathcal{D}}(\Gamma)_1 &\stackrel{\text{def}}{=} x_1 : \overrightarrow{\mathcal{D}}\tau_{11}, \dots, x_n : \overrightarrow{\mathcal{D}}\tau_{n1} && \text{(a cartesian typing context)} \\ \overrightarrow{\mathcal{D}}(\Gamma)_2 &\stackrel{\text{def}}{=} (\overrightarrow{\mathcal{D}}\tau_{12}[x_1/p] * \dots * \overrightarrow{\mathcal{D}}\tau_{n2}[x_n/p]) && \text{(a linear type)} \\ \overleftarrow{\mathcal{D}}(\Gamma)_1 &\stackrel{\text{def}}{=} x_1 : \overleftarrow{\mathcal{D}}(\tau_1)_1, \dots, x_n : \overleftarrow{\mathcal{D}}(\tau_n)_1 && \text{(a cartesian typing context)} \\ \overleftarrow{\mathcal{D}}(\Gamma)_2 &\stackrel{\text{def}}{=} (\overleftarrow{\mathcal{D}}(\tau_1)_2[x_1/p] * \dots * \overleftarrow{\mathcal{D}}(\tau_n)_2[x_n/p]) && \text{(a linear type)}. \end{aligned}$$

Our code transformations are well kinded in the sense that they translate a type $\Delta \vdash \tau : \text{type}$ of the source language into pairs of types of the target language:

$$\begin{aligned} \Delta \mid \cdot \vdash \overrightarrow{\mathcal{D}}(\tau)_1 : \text{type} \\ \Delta \mid p : \overrightarrow{\mathcal{D}}(\tau)_1 \vdash \overrightarrow{\mathcal{D}}(\tau)_2 : \text{ltype} \end{aligned}$$

$$\Delta \mid \cdot \vdash \overleftarrow{\mathcal{D}}(\tau)_1 : \text{type}$$

$$\Delta \mid p : \overleftarrow{\mathcal{D}}(\tau)_1 \vdash \overleftarrow{\mathcal{D}}(\tau)_2 : \text{ltype}.$$

Similarly, the functors $\overrightarrow{\mathcal{D}}(-) : \mathbf{Syn} \rightarrow \Sigma_{\mathbf{CSyn}} \mathbf{LSyn}$ and $\overleftarrow{\mathcal{D}}(-) : \mathbf{Syn} \rightarrow \Sigma_{\mathbf{CSyn}} \mathbf{LSyn}^{op}$ define for each term t of the source language and a list $\overline{\Gamma}$ of identifiers that contains at least the free identifiers of t :

- a term $\overrightarrow{\mathcal{D}}_{\overline{\Gamma}}(t)_1$ that represents the forward-mode primal computation associated with t ;
- a term $\overrightarrow{\mathcal{D}}_{\overline{\Gamma}}(t)_2$ that represents the forward-mode tangent computation associated with t ;
- a term $\overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(t)_1$ that represents the reverse-mode primal computation associated with t ;
- a term $\overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(t)_2$ that represents the reverse-mode cotangent computation associated with t .

These code transformations are well typed in the sense that a source language term t that is typed according to $\Delta \mid \Gamma \vdash t : \tau$ is translated into pairs of terms of the target language that are typed as follows:

$$\Delta \mid \overrightarrow{\mathcal{D}}(\Gamma)_1 \vdash \overrightarrow{\mathcal{D}}_{\overline{\Gamma}}(t)_1 : \overrightarrow{\mathcal{D}}(\tau)_1$$

$$\Delta \mid \overrightarrow{\mathcal{D}}(\Gamma)_1; \nu : \overrightarrow{\mathcal{D}}(\Gamma)_2 \vdash \overrightarrow{\mathcal{D}}_{\overline{\Gamma}}(t)_2 : \overrightarrow{\mathcal{D}}(\tau)_2[\overrightarrow{\mathcal{D}}_{\overline{\Gamma}}(t)_1/p]$$

$$\Delta \mid \overleftarrow{\mathcal{D}}(\Gamma)_1 \vdash \overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(t)_1 : \overleftarrow{\mathcal{D}}(\tau)_1$$

$$\Delta \mid \overleftarrow{\mathcal{D}}(\Gamma)_1; \nu : \overleftarrow{\mathcal{D}}(\tau)_2[\overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(t)_1/p] \vdash \overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(t)_2 : \overleftarrow{\mathcal{D}}(\Gamma)_2,$$

where $\overline{\Gamma}$ is the list of identifiers that occurs in Γ (that is, $\overline{x_1 : \tau_1, \dots, x_n : \tau_n} \stackrel{\text{def}}{=} x_1, \dots, x_n$).

However, as we noted already in Insight 1 of Section 2, we often want to share computation between the primal and (co)tangent values, for reasons of efficiency. Therefore, we focus instead on transforming a source language term $\Delta \mid \Gamma \vdash t : \tau$ into target language terms:

$$\Delta \mid \overrightarrow{\mathcal{D}}(\Gamma)_1 \vdash \overrightarrow{\mathcal{D}}_{\overline{\Gamma}}(t) : \Sigma p : \overrightarrow{\mathcal{D}}(\tau)_1. \overrightarrow{\mathcal{D}}(\Gamma)_2 \multimap \overrightarrow{\mathcal{D}}(\tau)_2$$

$$\Delta \mid \overleftarrow{\mathcal{D}}(\Gamma)_1 \vdash \overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(t) : \Sigma p : \overleftarrow{\mathcal{D}}(\tau)_1. \overleftarrow{\mathcal{D}}(\tau)_2 \multimap \overleftarrow{\mathcal{D}}(\Gamma)_2,$$

where $\overrightarrow{\mathcal{D}}_{\overline{\Gamma}}(t) \stackrel{\text{def}}{=} \langle \overrightarrow{\mathcal{D}}_{\overline{\Gamma}}(t)_1, \lambda \nu. \overrightarrow{\mathcal{D}}_{\overline{\Gamma}}(t)_2 \rangle$, and $\overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(t) \stackrel{\text{def}}{=} \langle \overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(t)_1, \lambda \nu. \overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(t)_2 \rangle$. While both representations of AD on programs are equivalent in terms of the $\beta\eta+$ -equational theory of the target language and therefore for any semantic and correctness purposes, they are meaningfully different in terms of efficiency. Indeed, we ensure that common subcomputations between the primals and (co)tangents are shared via let-bindings in $\overrightarrow{\mathcal{D}}_{\overline{\Gamma}}(t)$ and $\overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(t)$.

8.3 Code transformations of primitive types and operations

We have suitable terms (linear operations):

$$x_1 : \mathbf{real}^{n_1}, \dots, x_k : \mathbf{real}^{n_k} ; \nu : \mathbf{real}^{n_1} * \dots * \mathbf{real}^{n_k} \vdash \text{Dop}(x_1, \dots, x_k; \nu) : \mathbf{real}^m$$

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

to represent the forward- and reverse-mode derivatives of the primitive operations $\text{op} \in \text{Op}_{n_1, \dots, n_k}^m$. Using these, we define

$$\overrightarrow{\mathcal{D}} \mathbf{real}^n_1 \stackrel{\text{def}}{=} \mathbf{real}^n \qquad \overrightarrow{\mathcal{D}} \mathbf{real}^n_2 \stackrel{\text{def}}{=} \mathbf{real}^n$$

$$\overrightarrow{\mathcal{D}}_{\overline{\Gamma}}(\text{op}(t_1, \dots, t_k)) \stackrel{\text{def}}{=} \dots \qquad \text{let } \langle x_1, x'_1 \rangle = \overrightarrow{\mathcal{D}}_{\overline{\Gamma}}(t_1) \text{ in } \dots \text{let } \langle x_k, x'_k \rangle = \overrightarrow{\mathcal{D}}_{\overline{\Gamma}}(t_k) \text{ in}$$

$$\langle \text{op}(x_1, \dots, x_k), \underline{\lambda}v. \text{Dop}(x_1, \dots, x_n; \langle x'_1 \bullet v, \dots, x'_k \bullet v \rangle) \rangle$$

$$\overleftarrow{D}(\mathbf{real}^n)_1 \stackrel{\text{def}}{=} \mathbf{real}^n \qquad \overleftarrow{D}(\mathbf{real}^n)_2 \stackrel{\text{def}}{=} \underline{\underline{\mathbf{real}^n}}$$

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

For the AD transformations to be correct, it is important that these derivatives of language primitives are implemented correctly in the sense that

$$\llbracket x_1, \dots, x_k; y \vdash \text{Dop}(x_1, \dots, x_k; v) \rrbracket = D[\llbracket \text{op} \rrbracket] \qquad \llbracket x_1, \dots, x_k; v \vdash \text{Dop}^t(x_1, \dots, x_k; v) \rrbracket = D[\llbracket \text{op} \rrbracket]^t.$$

For example, for elementwise multiplication $(*) \in \text{Op}_{n,n}^n$, we need that

$$\begin{aligned} \llbracket D(*) (x_1, x_2; v) \rrbracket ((a_1, a_2), (b_1, b_2)) &= a_1 * b_2 + a_2 * b_1; \\ \llbracket D(*)^t (x_1, x_2; v) \rrbracket ((a_1, a_2), b) &= (a_2 * b, a_1 * b). \end{aligned}$$

By Corollary 15, the extension of the AD transformations \overrightarrow{D} and \overleftarrow{D} to the full source language are now canonically determined, as the unique $\mu\nu$ -polynomials-preserving bicartesian closed functors that extend the previous definitions.

8.4 Forward-mode CHAD definitions

We define the types of (forward-mode) primals $\mathcal{D}(\tau)_1$ and tangents $\overrightarrow{D}(\tau)_2$ associated with a type τ as follows:

$$\begin{aligned} \overrightarrow{D}(\mathbf{1})_1 &\stackrel{\text{def}}{=} \mathbf{1} \\ \overrightarrow{D}(\tau * \sigma)_1 &\stackrel{\text{def}}{=} \overrightarrow{D}(\tau)_1 * \overrightarrow{D}(\sigma)_1 \\ \overrightarrow{D}(\tau \rightarrow \sigma)_1 &\stackrel{\text{def}}{=} \Pi p : \overrightarrow{D}(\tau)_1. \Sigma p' : \overrightarrow{D}(\sigma)_1. \overrightarrow{D}(\tau)_2 \multimap \overrightarrow{D}(\sigma)_2 [p'/p] \\ \overrightarrow{D}(\{\ell_1 \tau_1 \mid \dots \mid \ell_n \tau_n\})_1 &\stackrel{\text{def}}{=} \{ \ell_1 \overrightarrow{D}(\tau_1)_1 \mid \dots \mid \ell_n \overrightarrow{D}(\tau_n)_1 \} \\ \overrightarrow{D}(\alpha)_1 &\stackrel{\text{def}}{=} \alpha \\ \overrightarrow{D}(\mu \alpha. \tau)_1 &\stackrel{\text{def}}{=} \mu \alpha. \overrightarrow{D}(\tau)_1 \\ \overrightarrow{D}(\nu \alpha. \tau)_1 &\stackrel{\text{def}}{=} \nu \alpha. \overrightarrow{D}(\tau)_1 \\ \overrightarrow{D}(\mathbf{1})_2 &\stackrel{\text{def}}{=} \underline{\mathbf{1}} \\ \overrightarrow{D}(\tau * \sigma)_2 &\stackrel{\text{def}}{=} \overrightarrow{D}(\tau)_2 [\mathbf{fst} p/p] * \overrightarrow{D}(\sigma)_2 [\mathbf{snd} p/p] \\ \overrightarrow{D}(\tau \rightarrow \sigma)_2 &\stackrel{\text{def}}{=} \Pi p' : \overrightarrow{D}(\tau)_1. \overrightarrow{D}(\sigma)_2 [\mathbf{fst} (p p')/p] \\ \overrightarrow{D}(\{\ell_1 \tau_1 \mid \dots \mid \ell_n \tau_n\})_2 &\stackrel{\text{def}}{=} \mathbf{case} p \mathbf{of} \{ \ell_1 p \rightarrow \overrightarrow{D}(\tau_1)_2 \mid \dots \mid \ell_n p \rightarrow \overrightarrow{D}(\tau_n)_2 \} \\ \overrightarrow{D}(\alpha)_2 &\stackrel{\text{def}}{=} \underline{\alpha} \\ \overrightarrow{D}(\mu \alpha. \tau)_2 &\stackrel{\text{def}}{=} \underline{\mu} \alpha. \overrightarrow{D}(\tau)_2 [\mathbf{fold} p \mathbf{with} y \rightarrow \overrightarrow{D}(\tau)_1 [y^{\mathbf{tr-roll}}/a] / p] \\ \overrightarrow{D}(\nu \alpha. \tau)_2 &\stackrel{\text{def}}{=} \underline{\nu} \alpha. \overrightarrow{D}(\tau)_2 [\mathbf{unroll} p/p] \end{aligned}$$

For programs t , we define their efficient CHAD transformation $\vec{D}_{\bar{\Gamma}}(t)$ as follows (and we list the less efficient transformations $\vec{D}_{\bar{\Gamma}}(t)_1$ and $\vec{D}_{\bar{\Gamma}}(t)_2$ that do not share computations between the primals and tangents in Appendix B):

$$\begin{aligned}
 \vec{D}_{\bar{\Gamma}}(x) &\stackrel{\text{def}}{=} \dots\dots\dots \langle x, \underline{\lambda}v. \mathbf{proj}_{\text{id}\mathbf{x}(x;\bar{\Gamma})} (v) \rangle \\
 \vec{D}_{\bar{\Gamma}}(\mathbf{let } x = t \mathbf{ in } s) &\stackrel{\text{def}}{=} \dots\dots\dots \mathbf{let } \langle x, x' \rangle = \vec{D}_{\bar{\Gamma}}(t) \mathbf{ in} \\
 &\quad \mathbf{let } \langle y, y' \rangle = \vec{D}_{\bar{\Gamma},x}(s) \mathbf{ in} \\
 &\quad \langle y, \underline{\lambda}v. y' \bullet \langle v, x' \bullet v \rangle \rangle \\
 \vec{D}_{\bar{\Gamma}}(\langle \rangle) &\stackrel{\text{def}}{=} \dots\dots\dots \langle \langle \rangle, \underline{\lambda}v. \langle \rangle \rangle \\
 \vec{D}_{\bar{\Gamma}}(\langle t, s \rangle) &\stackrel{\text{def}}{=} \dots\dots\dots \mathbf{let } \langle x, x' \rangle = \vec{D}_{\bar{\Gamma}}(t) \mathbf{ in} \\
 &\quad \mathbf{let } \langle y, y' \rangle = \vec{D}_{\bar{\Gamma}}(s) \mathbf{ in} \\
 &\quad \langle \langle x, y \rangle, \underline{\lambda}v. \langle x' \bullet v, y' \bullet v \rangle \rangle \\
 \vec{D}_{\bar{\Gamma}}(\mathbf{fst } t) &\stackrel{\text{def}}{=} \dots\dots\dots \mathbf{let } \langle x, x' \rangle = \vec{D}_{\bar{\Gamma}}(t) \mathbf{ in } \langle \mathbf{fst } x, \underline{\lambda}v. \mathbf{fst } (x' \bullet v) \rangle \\
 \vec{D}_{\bar{\Gamma}}(\mathbf{snd } t) &\stackrel{\text{def}}{=} \dots\dots\dots \mathbf{let } \langle x, x' \rangle = \vec{D}_{\bar{\Gamma}}(t) \mathbf{ in } \langle \mathbf{snd } x, \underline{\lambda}v. \mathbf{snd } (x' \bullet v) \rangle \\
 \vec{D}_{\bar{\Gamma}}(\lambda x. t) &\stackrel{\text{def}}{=} \dots\dots\dots \mathbf{let } y = \lambda x. \vec{D}_{\bar{\Gamma},x}(t) \mathbf{ in} \\
 &\quad \langle \lambda x. \mathbf{let } \langle z, z' \rangle = y x \mathbf{ in } \langle z, \underline{\lambda}v. z' \bullet \langle \mathbf{0}, v \rangle \rangle, \underline{\lambda}v. \lambda x. (\mathbf{snd } (y x)) \bullet \langle v, \mathbf{0} \rangle \rangle \\
 \vec{D}_{\bar{\Gamma}}(t s) &\stackrel{\text{def}}{=} \dots\dots\dots \mathbf{let } \langle x, x'_{\text{ctx}} \rangle = \vec{D}_{\bar{\Gamma}}(t) \mathbf{ in } \mathbf{let } \langle y, y' \rangle = \vec{D}_{\bar{\Gamma}}(s) \mathbf{ in } \mathbf{let } \langle z, z'_{\text{arg}} \rangle = x y \mathbf{ in} \\
 &\quad \langle z, \underline{\lambda}v. (x'_{\text{ctx}} \bullet v) y + x'_{\text{arg}} \bullet (y' \bullet v) \rangle \\
 \vec{D}_{\bar{\Gamma}}(\ell t) &\stackrel{\text{def}}{=} \dots\dots\dots \mathbf{let } \langle x, x' \rangle = \vec{D}_{\bar{\Gamma}}(t) \mathbf{ in } \langle \ell x, x' \rangle \\
 \vec{D}_{\bar{\Gamma}}(\mathbf{case } t \mathbf{ of } \{ \ell_1 x_1 \rightarrow s_1 \mid \dots \mid \ell_n x_n \rightarrow s_n \}) &\stackrel{\text{def}}{=} \dots\dots\dots \mathbf{let } \langle y, y' \rangle = \vec{D}_{\bar{\Gamma}}(t) \mathbf{ in} \\
 &\quad \mathbf{case } y \mathbf{ of } \{ \ell_1 x_1 \rightarrow \\
 &\quad \quad \mathbf{let } \langle z_1, z'_1 \rangle = \vec{D}_{\bar{\Gamma},x_1}(s_1) \mathbf{ in} \\
 &\quad \quad \langle z_1, \underline{\lambda}v. z'_1 \bullet \langle v, (\mathbf{let } y = \ell_1 x_1 \mathbf{ in } y') \bullet v \rangle \rangle \\
 &\quad \quad \mid \dots \mid \\
 &\quad \quad \ell_n x_n \rightarrow \\
 &\quad \quad \mathbf{let } \langle z_n, z'_n \rangle = \vec{D}_{\bar{\Gamma},x_n}(s_n) \mathbf{ in} \\
 &\quad \quad \langle z_n, \underline{\lambda}v. z'_n \bullet \langle v, (\mathbf{let } y = \ell_n x_n \mathbf{ in } y') \bullet v \rangle \rangle \} \\
 \vec{D}_{\bar{\Gamma}}(\mathbf{roll } t) &\stackrel{\text{def}}{=} \dots\dots\dots \mathbf{let } \langle x, x' \rangle = \vec{D}_{\bar{\Gamma}}(t) \mathbf{ in } \langle \mathbf{roll } x, \underline{\lambda}v. \mathbf{roll } (x' \bullet v) \rangle \\
 \vec{D}_{\bar{\Gamma}}(\mathbf{fold } t \mathbf{ with } x \rightarrow s) &\stackrel{\text{def}}{=} \dots\dots\dots \mathbf{let } \langle y, y' \rangle = \vec{D}_{\bar{\Gamma}}(t) \mathbf{ in} \\
 &\quad \mathbf{let } z = \lambda x. \vec{D}_x(s) \mathbf{ in}
 \end{aligned}$$

$$\begin{aligned}
 & \langle \mathbf{fold} \ y \ \mathbf{with} \ x \rightarrow \mathbf{fst} \ (z \ x), \\
 & \underline{\lambda}v. \mathbf{fold} \ y' \bullet v \ \mathbf{with} \ v \rightarrow \\
 & \mathbf{let} \ x = \mathbf{fold} \ y \ \mathbf{with} \ x \rightarrow \overrightarrow{\mathcal{D}}(\tau)_1[x^{\mathbf{fst}}(z \ x)/\alpha] \ \mathbf{in} \ (\mathbf{snd} \ (z \ x)) \bullet v) \\
 \overrightarrow{\mathcal{D}}_{\overline{\Gamma}}(\mathbf{unroll} \ t) \stackrel{\text{def}}{=} \dots & \quad \mathbf{let} \ \langle x, x' \rangle = \overrightarrow{\mathcal{D}}_{\overline{\Gamma}}(t) \ \mathbf{in} \ (\mathbf{unroll} \ x, \underline{\lambda}v. \mathbf{unroll} \ (x' \bullet v)) \\
 \overrightarrow{\mathcal{D}}_{\overline{\Gamma}}(\mathbf{gen} \ \mathbf{from} \ t \ \mathbf{with} \ x \rightarrow s) \stackrel{\text{def}}{=} \dots & \quad \mathbf{let} \ \langle y, y' \rangle = \overrightarrow{\mathcal{D}}_{\overline{\Gamma}}(t) \ \mathbf{in} \\
 & \mathbf{let} \ z = \lambda x. \overrightarrow{\mathcal{D}}_x(s) \ \mathbf{in} \\
 & \langle \mathbf{gen} \ \mathbf{from} \ y \ \mathbf{with} \ x \rightarrow \mathbf{fst} \ (z \ x), \\
 & \underline{\lambda}v. \mathbf{gen} \ \mathbf{from} \ y' \bullet v \ \mathbf{with} \ v \rightarrow \bullet(\mathbf{snd} \ (z \ y)) \bullet v)
 \end{aligned}$$

8.5 Reverse-mode CHAD definitions

We define the types of (reverse-mode) primals $\overleftarrow{\mathcal{D}}(\tau)_1$ and cotangents $\overleftarrow{\mathcal{D}}(\tau)_2$ associated with a type τ as follows:

$$\begin{aligned}
 \overleftarrow{\mathcal{D}}(\mathbf{1})_1 & \stackrel{\text{def}}{=} \mathbf{1} \\
 \overleftarrow{\mathcal{D}}(\tau * \sigma)_1 & \stackrel{\text{def}}{=} \overleftarrow{\mathcal{D}}(\tau)_1 * \overleftarrow{\mathcal{D}}(\sigma)_1 \\
 \overleftarrow{\mathcal{D}}(\tau \rightarrow \sigma)_1 & \stackrel{\text{def}}{=} \Pi p : \overleftarrow{\mathcal{D}}(\tau)_1. \Sigma p' : \overleftarrow{\mathcal{D}}(\sigma)_1. \overleftarrow{\mathcal{D}}(\sigma)_2[p'/p] \multimap \overleftarrow{\mathcal{D}}(\tau)_2 \\
 \overleftarrow{\mathcal{D}}(\{\ell_1 \tau_1 \mid \dots \mid \ell_n \tau_n\})_1 & \stackrel{\text{def}}{=} \{\ell_1 \overleftarrow{\mathcal{D}}(\tau_1)_1 \mid \dots \mid \ell_n \overleftarrow{\mathcal{D}}(\tau_n)_1\} \\
 \overleftarrow{\mathcal{D}}(\alpha)_1 & \stackrel{\text{def}}{=} \alpha \\
 \overleftarrow{\mathcal{D}}(\mu \alpha. (\tau))_1 & \stackrel{\text{def}}{=} \mu \alpha. \overleftarrow{\mathcal{D}}(\tau)_1 \\
 \overleftarrow{\mathcal{D}}(\nu \alpha. (\tau))_1 & \stackrel{\text{def}}{=} \nu \alpha. \overleftarrow{\mathcal{D}}(\tau)_1 \\
 \\
 \overleftarrow{\mathcal{D}}(\mathbf{1})_2 & \stackrel{\text{def}}{=} \underline{\mathbf{1}} \\
 \overleftarrow{\mathcal{D}}(\tau * \sigma)_2 & \stackrel{\text{def}}{=} \overleftarrow{\mathcal{D}}(\tau)_2[\mathbf{fst} \ p/p] * \overleftarrow{\mathcal{D}}(\sigma)_2[\mathbf{snd} \ p/p] \\
 \overleftarrow{\mathcal{D}}(\tau \rightarrow \sigma)_2 & \stackrel{\text{def}}{=} \Sigma p' : \overleftarrow{\mathcal{D}}(\tau)_1. \overleftarrow{\mathcal{D}}(\sigma)_2[\mathbf{fst} \ (p \ p')/p] \\
 \overleftarrow{\mathcal{D}}(\{\ell_1 \tau_1 \mid \dots \mid \ell_n \tau_n\})_2 & \stackrel{\text{def}}{=} \mathbf{case} \ p \ \mathbf{of} \ \{\ell_1 p \rightarrow \overleftarrow{\mathcal{D}}(\tau_1)_2 \mid \dots \mid \ell_n p \rightarrow \overleftarrow{\mathcal{D}}(\tau_n)_2\} \\
 \overleftarrow{\mathcal{D}}(\alpha)_2 & \stackrel{\text{def}}{=} \alpha \\
 \overleftarrow{\mathcal{D}}(\mu \alpha. (\tau))_2 & \stackrel{\text{def}}{=} \underline{\nu} \alpha. \overleftarrow{\mathcal{D}}(\tau)_2[\mathbf{fold} \ p \ \mathbf{with} \ y \rightarrow \overleftarrow{\mathcal{D}}(\tau)_1[y^{\mathbf{fst}}(y/\alpha)/p]] \\
 \overleftarrow{\mathcal{D}}(\nu \alpha. \tau)_2 & \stackrel{\text{def}}{=} \underline{\mu} \alpha. \overleftarrow{\mathcal{D}}(\tau)_2[\mathbf{unroll} \ p/p]
 \end{aligned}$$

For programs t , we define their efficient CHAD transformation $\overrightarrow{\mathcal{D}}_{\overline{\Gamma}}(t)$ as follows (and we list the less efficient transformations $\overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(t)_1$ and $\overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(t)_2$ that do not share computation between the primals and cotangents in Appendix B):

$$\overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(x) \stackrel{\text{def}}{=} \dots \quad \langle x, \underline{\lambda}v. \mathbf{coproj}_{\mathbf{idx}(x; \overline{\Gamma})} \ (v) \rangle$$

$$\overleftarrow{\mathcal{D}}_{\Gamma}(\mathbf{let} \ x = t \ \mathbf{in} \ s) \stackrel{\text{def}}{=} \dots \quad \mathbf{let} \ \langle x, x' \rangle = \overleftarrow{\mathcal{D}}_{\Gamma}(t) \ \mathbf{in}$$

$$\mathbf{let} \ \langle y, y' \rangle = \overleftarrow{\mathcal{D}}_{\Gamma, x}(s) \ \mathbf{in}$$

$$\langle y, \underline{\lambda}v. \mathbf{let} \ v = y' \bullet v \ \mathbf{in} \ \mathbf{fst} \ v + x' \bullet (\mathbf{snd} \ v) \rangle$$

$$\overleftarrow{\mathcal{D}}_{\Gamma}(\langle \rangle) \stackrel{\text{def}}{=} \dots \quad \langle \langle \rangle, \underline{\lambda}v. 0 \rangle$$

$$\overleftarrow{\mathcal{D}}_{\Gamma}(\langle t, s \rangle) \stackrel{\text{def}}{=} \dots \quad \mathbf{let} \ \langle x, x' \rangle = \overleftarrow{\mathcal{D}}_{\Gamma}(t) \ \mathbf{in}$$

$$\mathbf{let} \ \langle y, y' \rangle = \overleftarrow{\mathcal{D}}_{\Gamma, s} \ \mathbf{in}$$

$$\langle \langle x, y \rangle, \underline{\lambda}v. x' \bullet (\mathbf{fst} \ v) \rangle + y' \bullet (\mathbf{snd} \ v)$$

$$\overleftarrow{\mathcal{D}}_{\Gamma}(\mathbf{fst} \ t) \stackrel{\text{def}}{=} \dots \quad \mathbf{let} \ \langle x, x' \rangle = \overleftarrow{\mathcal{D}}_{\Gamma}(t) \ \mathbf{in} \ \langle \mathbf{fst} \ x, \underline{\lambda}v. x' \bullet \langle v, 0 \rangle \rangle$$

$$\overleftarrow{\mathcal{D}}_{\Gamma}(\mathbf{snd} \ t) \stackrel{\text{def}}{=} \dots \quad \mathbf{let} \ \langle x, x' \rangle = \overleftarrow{\mathcal{D}}_{\Gamma}(t) \ \mathbf{in} \ \langle \mathbf{snd} \ x, \underline{\lambda}v. x' \bullet \langle 0, v \rangle \rangle$$

$$\overleftarrow{\mathcal{D}}_{\Gamma}(\lambda x. t) \stackrel{\text{def}}{=} \dots \quad \mathbf{let} \ y = \lambda x. \overleftarrow{\mathcal{D}}_{\Gamma, x}(t) \ \mathbf{in}$$

$$\langle \lambda x. \mathbf{let} \ \langle z, z' \rangle = y \ x \ \mathbf{in} \ \langle z, \underline{\lambda}v. \mathbf{snd} \ (z' \bullet v) \rangle,$$

$$\underline{\lambda}v. \mathbf{case} \ v \ \mathbf{of} \ !x \otimes v \rightarrow \mathbf{fst} \ (\langle \mathbf{snd} \ (y \ x) \rangle \bullet v) \rangle$$

$$\overleftarrow{\mathcal{D}}_{\Gamma}(t \ s) \stackrel{\text{def}}{=} \dots \quad \mathbf{let} \ \langle x, x'_{\text{ctx}} \rangle = \overleftarrow{\mathcal{D}}_{\Gamma}(t) \ \mathbf{in} \ \mathbf{let} \ \langle y, y' \rangle = \overleftarrow{\mathcal{D}}_{\Gamma, s} \ \mathbf{in} \ \mathbf{let} \ \langle z, x'_{\text{arg}} \rangle = x \ y \ \mathbf{in}$$

$$\langle z, \underline{\lambda}v. x'_{\text{ctx}} \bullet (!y \otimes v) + y' \bullet (x'_{\text{arg}} \bullet v) \rangle$$

$$\overleftarrow{\mathcal{D}}_{\Gamma}(\ell t) \stackrel{\text{def}}{=} \dots \quad \mathbf{let} \ \langle x, x' \rangle = \overleftarrow{\mathcal{D}}_{\Gamma}(t) \ \mathbf{in} \ \langle \ell x, x' \rangle$$

$$\overleftarrow{\mathcal{D}}_{\Gamma}(\mathbf{case} \ t \ \mathbf{of} \ \{\ell_1 x_1 \rightarrow s_1 \mid \dots \mid \ell_n x_n \rightarrow s_n\}) \stackrel{\text{def}}{=} \dots \quad \mathbf{let} \ \langle y, y' \rangle = \overleftarrow{\mathcal{D}}_{\Gamma}(t) \ \mathbf{in}$$

$$\mathbf{case} \ \mathbf{yof} \ \{\ell_1 x_1 \rightarrow$$

$$\mathbf{let} \ \langle z_1, z'_1 \rangle = \overleftarrow{\mathcal{D}}_{\Gamma, x_1}(s_1) \ \mathbf{in}$$

$$\langle z_1, \underline{\lambda}v. \mathbf{let} \ v = z'_1 \bullet v \ \mathbf{in} \ \mathbf{fst} \ v +$$

$$\quad (\mathbf{let} \ y = \ell_1 x_1 \ \mathbf{in} \ y') \bullet (\mathbf{snd} \ v) \rangle$$

$$\mid \dots \mid$$

$$\ell_n x_n \rightarrow \mathbf{let} \ \langle z_n, z'_n \rangle = \overleftarrow{\mathcal{D}}_{\Gamma, x_n}(s_n) \ \mathbf{in}$$

$$\langle z_n, \underline{\lambda}v. \mathbf{let} \ v = z'_n \bullet v \ \mathbf{in} \ \mathbf{fst} \ v +$$

$$\quad (\mathbf{let} \ y = \ell_n x_n \ \mathbf{in} \ y') \bullet (\mathbf{snd} \ v) \rangle \}$$

$$\overleftarrow{\mathcal{D}}_{\Gamma}(\mathbf{roll} \ t) \stackrel{\text{def}}{=} \dots \quad \mathbf{let} \ \langle x, x' \rangle = \overleftarrow{\mathcal{D}}_{\Gamma}(t) \ \mathbf{in} \ \langle \mathbf{roll} \ x, \underline{\lambda}v. x' \bullet (\mathbf{unroll} \ v) \rangle$$

$$\overleftarrow{\mathcal{D}}_{\Gamma}(\mathbf{fold} \ t \ \mathbf{with} \ x \rightarrow s) \stackrel{\text{def}}{=} \dots \quad \mathbf{let} \ \langle y, y' \rangle = \overleftarrow{\mathcal{D}}_{\Gamma}(t) \ \mathbf{in}$$

$$\mathbf{let} \ z = \lambda x. \overleftarrow{\mathcal{D}}_x(s) \ \mathbf{in}$$

$$\langle \mathbf{fold} \ y \ \mathbf{with} \ x \rightarrow \mathbf{fst} \ (z \ x)$$

$$\quad , \underline{\lambda}v. y' \bullet \mathbf{gen} \ \mathbf{from} \ v \ \mathbf{with} \ v \rightarrow$$

$$\begin{aligned}
 & \text{let } x = \text{fold } y \text{ with } x \rightarrow \overrightarrow{\mathcal{D}}(\tau)_1[x^{\dagger}\text{fst}(zx)/\alpha] \text{ in } (\text{snd}(zx)) \bullet v \\
 \overleftarrow{\mathcal{D}}_{\Gamma}(\text{unroll } t) & \stackrel{\text{def}}{=} \dots \quad \text{let } \langle x, x' \rangle = \overleftarrow{\mathcal{D}}_{\Gamma}(t) \text{ in } (\text{unroll } x, \underline{\lambda}v.x' \bullet (\text{roll } v)) \\
 \overleftarrow{\mathcal{D}}_{\Gamma}(\text{gen from } t \text{ with } x \rightarrow s) & \stackrel{\text{def}}{=} \dots \quad \text{let } \langle y, y' \rangle = \overleftarrow{\mathcal{D}}_{\Gamma}(t) \text{ in} \\
 & \text{let } z = \lambda x. \overleftarrow{\mathcal{D}}_x(s) \text{ in} \\
 & \langle \text{gen from } y \text{ with } x \rightarrow \text{fst}(zx) \\
 & \quad , \underline{\lambda}v.y' \bullet \text{fold } v \text{ with } v \rightarrow (\text{snd}(zy)) \bullet v \rangle
 \end{aligned}$$

9. Concrete Models

In order to proceed with our correctness proof of AD, we need to establish the semantics of the program transformation in our setting. In this section, we construct denotational semantics for the target language.

9.1 Locally presentable categories and μv -polynomials

We show that any cartesian closed locally presentable category yields a concrete model for the source language. The only step needed to establish this fact is to prove that locally presentable categories have μv -polynomials, cf. Santocanale (2002, Theorem 3.7). We establish this result below. We refer the reader to Adámek and Rosický (1994) and Bird (1984) for basics on locally presentable categories.

The first fact to recall is that locally presentable categories are complete (and cocomplete by definition): see, for instance, Adámek and Rosický (1994, p. 45). Moreover:

Lemma 70. *Let \mathcal{A}, \mathcal{B} be locally presentable categories.*

- (A) *A functor $G : \mathcal{A} \rightarrow \mathcal{B}$ has a left adjoint if and only if G is accessible and preserves limits.*
- (B) *A functor $F : \mathcal{B} \rightarrow \mathcal{A}$ has a right adjoint if and only if F preserves colimits.*

Proof. (A) is Adámek and Rosický (1994, Theorem 1.66).

Recall that every locally presentable is co-well-powered; see Adámek and Rosický (1994, Theorem 1.58). By the special adjoint functor theorem (Mac Lane 1971, p. 129), we get that (B) holds. □

Lemma 71. *Every accessible endofunctor on a locally presentable category has an initial algebra and a terminal coalgebra.*

Proof. Every accessible endofunctor on a locally presentable category has an initial algebra since we construct the initial algebra via the colimit of the chain $0 \rightarrow E(0) \rightarrow \dots$; see Adámek and Koubek (1979).

If \mathcal{A} is a locally presentable category, given an endofunctor $E : \mathcal{A} \rightarrow \mathcal{A}$, we have that $E\text{-CoAlg}$ is locally presentable. Since the forgetful functor $E\text{-CoAlg} \rightarrow \mathcal{A}$ is a functor between locally presentable categories that creates colimits, we have that it has a right adjoint R . Therefore, $R(\mathbb{1})$ is the terminal object of $E\text{-CoAlg}$ (terminal coalgebra of E); see Barr (1993). □

Proposition 72. *If \mathcal{D} is locally presentable then \mathcal{D} has μv -polynomials.*

Proof. The terminal category $\mathbb{1}$ is a locally presentable category and, if \mathcal{D}' and \mathcal{D}'' are locally presentable categories, then $\mathcal{D}' \times \mathcal{D}''$ is locally presentable as well. Therefore, all the objects of $\mu\nu\text{Poly}_{\mathcal{D}}$ are locally presentable.

Given locally presentable categories $\mathcal{D}', \mathcal{D}''$, the projections $\pi_1 : \mathcal{D}' \times \mathcal{D}'' \rightarrow \mathcal{D}'$ and $\pi_2 : \mathcal{D}' \times \mathcal{D}'' \rightarrow \mathcal{D}''$ have right (and left) adjoints and, therefore, are accessible.

Moreover, given locally presentable categories $\mathcal{D}', \mathcal{D}'', \mathcal{D}'''$, if $E : \mathcal{D}' \rightarrow \mathcal{D}''$ and $J : \mathcal{D}' \rightarrow \mathcal{D}'''$ are accessible functors, then so is the induced functor $(E, J) : \mathcal{D}' \rightarrow \mathcal{D}'' \times \mathcal{D}'''$.

Furthermore, $\times : \mathcal{D} \times \mathcal{D} \rightarrow \mathcal{D}$ and $\sqcup : \mathcal{D} \times \mathcal{D} \rightarrow \mathcal{D}$ have, respectively, a left adjoint and a right adjoint. Therefore, they are accessible.

Finally, by Santocanale (2002, Proposition 3.8), assuming their existence, μH and νH are accessible whenever $H : \mathcal{D}' \times \mathcal{D} \rightarrow \mathcal{D}$ is accessible and \mathcal{D}' is locally presentable.

This completes the proof that all morphisms of $\mu\nu\text{Poly}_{\mathcal{D}}$ are accessible. Hence, by Lemma 71, we have that all endofunctors in $\mu\nu\text{Poly}_{\mathcal{D}}$ have initial algebras and terminal coalgebras. Therefore, \mathcal{D} has $\mu\nu$ -polynomials. \square

Remark 73 (Duality). Let \mathcal{D} be a category. By a well-known result by Gabriel–Ulmer (Gabriel and Ulmer 1971, 7.13), \mathcal{D} and \mathcal{D}^{op} are locally presentable if, and only if, \mathcal{D} is a complete lattice. Therefore, in general, the property of being locally presentable is not self-dual.

As remarked in Remark 7, the property of having $\mu\nu$ -polynomials is self-dual. Hence, by Proposition 72, we have that, whenever \mathcal{D}^{op} is locally presentable, \mathcal{D} has $\mu\nu$ -polynomials.

9.2 Li, FLi, and Fam(Li)

Henceforth, we assume that \mathbf{Li} is a locally presentable category with biproducts $(+, 0)$ that is monadic over \mathbf{Set} . The main examples that we have in mind are the category of real vector spaces $\mathbf{Li} = \mathbf{Vect}$ and the category of commutative monoids $\mathbf{Li} = \mathbf{CMon}$.

We consider the indexed category:

$$\begin{aligned} \mathbf{FLi} : \mathbf{Set}^{\text{op}} &\rightarrow \mathbf{Cat} & (57) \\ X &\mapsto \mathbf{Cat}[X, \mathbf{Li}] = \mathbf{Li}^X \\ f : X \rightarrow Y &\mapsto \mathbf{Li}^f = \mathbf{Cat}[f, \mathbf{Li}] : \mathbf{Li}^Y \rightarrow \mathbf{Li}^X \end{aligned}$$

defined by the composition:

$$\mathbf{Set}^{\text{op}} \rightarrow \mathbf{Cat}^{\text{op}} \xrightarrow{\mathbf{Cat}[-, \mathbf{Li}]} \mathbf{Cat} \tag{58}$$

in which $\mathbf{Cat}[-, \mathbf{Li}] = \mathbf{Li}^{(-)}$ is the exponential (internal hom) in \mathbf{Cat} . We have that

$$\Sigma_{\mathbf{Set}} \mathbf{FLi} \cong \mathbf{Fam}(\mathbf{Li}), \quad \Sigma_{\mathbf{Set}} \mathbf{FLi}^{\text{op}} \cong \mathbf{Fam}(\mathbf{Li}^{\text{op}}) \tag{59}$$

where $\mathbf{Fam}(\mathbf{Li})$ and $\mathbf{Fam}(\mathbf{Li}^{\text{op}})$ are, respectively, the free cocompletion under coproducts of \mathbf{Li} and of \mathbf{Li}^{op} . We refer the reader, for instance, to Adámek and Rosický (2020, Section 2) and Borceux and Janelidze (2001, Chapter 6) for basic facts about free cocompletion under coproducts.

We have the following basic straightforward properties about $\mathbf{Fam}(\mathbf{Li})$:

Proposition 74. *Let \mathcal{D} be a category with biproducts $(+, 0)$. If \mathcal{D} has (infinite) products, $\mathbf{Fam}(\mathcal{D})$ is cartesian closed. Coadually, if \mathcal{D} has (infinite) coproducts, $\mathbf{Fam}(\mathcal{D}^{\text{op}})$ is cartesian closed.*

Proof. Namely, given families of objects $\mathcal{Y} : Y \rightarrow \mathcal{D}$, $\mathcal{Z} : Z \rightarrow \mathcal{D}$, we define

$$\mathcal{Y}\mathcal{Z} : \mathbf{Fam}(\mathcal{D})((Y, \mathcal{Y}), (Z, \mathcal{Z})) \rightarrow \mathcal{D} \tag{60}$$

$$(g : Y \rightarrow Z, (\alpha_y : \mathcal{Y}(y) \rightarrow \mathcal{Z}(g(y)))_{y \in Y}) \mapsto \prod_{y \in Y} \mathcal{Z}(g(y)) \tag{61}$$

$$\mathcal{Y}\mathcal{Z}^t : \mathbf{Fam}(\mathcal{D}^{\text{op}})((Y, \mathcal{Y}), (Z, \mathcal{Z})) \rightarrow \mathcal{D} \tag{62}$$

$$(g : Y \rightarrow Z, (\alpha_y : \mathcal{Z}(g(y)) \rightarrow \mathcal{Y}(y))_{y \in Y}) \mapsto \coprod_{y \in Y} \mathcal{Z}(g(y))$$

The pair $(\mathbf{Fam}(\mathcal{D})((Y, \mathcal{Y}), (Z, \mathcal{Z})), \mathcal{Y}\mathcal{Z})$ is the exponential $(Y, \mathcal{Y}) \Rightarrow (Z, \mathcal{Z})$ in $\mathbf{Fam}(\mathcal{D})$, provided that \mathcal{D} has products.

Codually, $(\mathbf{Fam}(\mathcal{D})((Y, \mathcal{Y}), (Z, \mathcal{Z})), \mathcal{Y}\mathcal{Z}^t)$ is the exponential $(Y, \mathcal{Y}) \Rightarrow (Z, \mathcal{Z})$ in $\mathbf{Fam}(\mathcal{D}^{\text{op}})$, provided that \mathcal{D} has coproducts. □

Proposition 75. *$\mathbf{Fam}(\mathcal{D})$ is locally presentable, whenever \mathcal{D} is locally presentable.*

Proof. Since \mathcal{D} is cocomplete, $\mathbf{Fam}(\mathcal{D})$ is cocomplete (see Lemma 80). Moreover, it is clear that the indexed category defined by $X \mapsto \mathbf{Cat}[X, \mathcal{D}]$ satisfies the conditions of Makkai and Paré (1989, Definition 5.3.1), since:

- (1) for each $X \in \mathbf{Set}$, $\mathbf{Cat}[X, \mathcal{D}] = \mathcal{D}^X$ is locally presentable and, hence, accessible;
- (2) for any function f , $\mathbf{Cat}[X, \mathcal{D}]$ is accessible by Lemma 70, since it has a left adjoint given by the left Kan extension lan_f ; see (64);
- (3) \mathbf{Set} is locally presentable;
- (4) $X \mapsto \mathbf{Cat}[X, \mathcal{D}]$ preserves any limit of \mathbf{Set}^{op} .

Therefore, $\mathbf{Fam}(\mathcal{D})$ is accessible by Makkai and Paré (1989, Theorem 5.3.4). This completes the proof that $\mathbf{Fam}(\mathcal{D})$ is locally presentable. □

As a consequence, we have that:

Corollary 76. *$\mathbf{Fam}(\mathbf{Li})$ is cartesian closed and locally presentable and, hence, has $\mu\nu$ -polynomials.*

The results proven above do not guarantee that $\mathbf{Fam}(\mathbf{Li}^{\text{op}})$ has $\mu\nu$ -polynomials, since $\mathbf{Fam}(\mathbf{Li}^{\text{op}})$ is not, generally, locally presentable. However, in 9.3, we show that \mathbf{FLi} yields a model for the target language and, hence, $\mathbf{Fam}(\mathbf{Li})$ and $\mathbf{Fam}(\mathbf{Li}^{\text{op}})$ have $\mu\nu$ -polynomials (and are cartesian closed).

9.3 FLi is a Σ -bimodel for inductive and coinductive types

We establish that $\mathbf{FLi} : \mathbf{Set}^{\text{op}} \rightarrow \mathbf{Cat}$ yields a model for the target language in Corollary 78. By the results of Section 6, this provides proof that $\Sigma_{\mathbf{Set}}\mathbf{FLi} \cong \mathbf{Fam}(\mathbf{Li})$ and $\Sigma_{\mathbf{Set}}\mathbf{FLi}^{\text{op}} \cong \mathbf{Fam}(\mathbf{Li}^{\text{op}})$ are bicartesian categories with $\mu\nu$ -polynomials by Corollary 66. We start by proving that \mathbf{FLi} is a Σ -bimodel for inductive and coinductive types.

Since \mathbf{Set} is locally presentable, \mathbf{Set} has $\mu\nu$ -polynomials by Proposition 72. Moreover, since \mathbf{Li} is complete and cocomplete, $\mathbf{FLi}(X) = \mathbf{Li}^X$ is complete and cocomplete as well; namely, the limits and colimits are constructed pointwise. In particular, $\mathbf{FLi}(X) = \mathbf{Li}^X$ has biproducts (also constructed pointwise) $(+, 0)$.

It should be noted that, for any function $f : X \rightarrow Y$ in \mathbf{Set} , we have that

$$\mathbf{Li}^f = \mathbf{FLi}(f) : \mathbf{Cat}[Y, \mathbf{Li}] \rightarrow \mathbf{Cat}[X, \mathbf{Li}] \tag{63}$$

has a (fully faithful) left adjoint and a (fully faithful) right adjoint, given by the left and right Kan extensions respectively;⁷ namely, for each $\mathcal{X} : X \rightarrow \mathbf{Li}$,

$$\text{ran}_f \mathcal{X}(x) = \prod_{i \in f^{-1}(x)} \mathcal{X}(i), \quad \text{lan}_f \mathcal{X}(x) = \prod_{i \in f^{-1}(x)} \mathcal{X}(i). \tag{64}$$

Therefore, we can conclude that: (1) $\mathbf{FLi}(f)$ preserves limits, colimits and, consequently, biproducts; (2) $\mathbf{FLi}(f)$ preserves initial algebras and terminal coalgebras by Theorem 108. Furthermore, $\mathbf{FLi}(f)$ strictly preserves biproducts (and the zero object), initial algebras and terminal coalgebras, provided that \mathbf{Li} has chosen ones.

Finally, it is clear that we have the isomorphism:

$$\begin{aligned} \mathbf{FLi}(X \sqcup Y) &= \mathbf{Cat}[X \sqcup Y, \mathbf{Li}] \\ &\cong \mathbf{Cat}[X, \mathbf{Li}] \times \mathbf{Cat}[Y, \mathbf{Li}] \\ &= \mathbf{FLi}(X) \times \mathbf{FLi}(Y) \end{aligned}$$

and, hence, \mathbf{FLi} is extensive. Indeed, we have

$$\mathcal{S}^{(X,Y)} : \mathbf{FLi}(X) \times \mathbf{FLi}(Y) \rightarrow \mathbf{FLi}(X \sqcup Y) \tag{65}$$

in which $\mathcal{S}^{(X,Y)}(\mathcal{X}, \mathcal{Y})(i) = \mathcal{X}(i)$ if $i \in X$ and $\mathcal{S}^{(X,Y)}(\mathcal{X}, \mathcal{Y})(j) = \mathcal{Y}(j)$ if $j \in Y$.

Theorem 77. *The strictly indexed category \mathbf{FLi} is a Σ -bimodel for inductive and coinductive types. Therefore, $\Sigma_{\mathbf{Set}}\mathbf{FLi}$ and $\Sigma_{\mathbf{Set}}\mathbf{FLi}^{\text{op}}$ have $\mu\nu$ -polynomials.*

Proof. It only remains to prove that all the endomorphisms in $\mu\nu\text{Poly}_{\mathbf{FLi}}$ have initial algebras and terminal coalgebras. In order to do so, by Lemma 71, it is enough to prove that $\mu\nu\text{Poly}_{\mathbf{FLi}}$ is a subcategory of the category of locally presentable categories and accessible functors between them.

The subcategory of locally presentable functors and accessible functors is closed under products. That is to say, if $\mathcal{D}, \mathcal{D}'$ are locally presentable categories and E, J are accessible functors between locally presentable categories, we get that $\mathbb{1}, \mathcal{D} \times \mathcal{D}'$ are locally presentable categories, (E, J) is accessible, and the projections are accessible (since they have right adjoints).

Moreover, \mathbf{Li}^X is locally presentable for any set X since \mathbf{Li} is locally presentable. Also, since the biproduct $+$: $\mathbf{Li}^X \times \mathbf{Li}^X \rightarrow \mathbf{Li}^X$ has a right adjoint, it is accessible. Furthermore, since it has a right adjoint, we get that $\mathbf{Li}(f)$ is accessible for any function $f : X \rightarrow Y$.

Finally, by Santocanale (2002, Proposition 3.8), assuming their existence, μh and νh are accessible whenever $h : \mathcal{D}' \times \mathcal{D} \rightarrow \mathcal{D}$ is accessible and $\mathcal{D}', \mathcal{D}$ are locally presentable categories.

Since isomorphisms between locally presentable categories are accessible, this completes the proof that all functors in $\mu\nu\text{Poly}_{\mathbf{FLi}}$ are accessible functors between locally presentable categories.

Therefore, any endomorphism in $\mu\nu\text{Poly}_{\mathbf{FLi}}$ has initial algebra and terminal coalgebra by Lemma 71. This completes the proof. □

9.4 FLi is a Σ -bimodel for function types

We consider the cartesian dependent type theory $\mathbf{FSet} : \mathbf{Set}^{\text{op}} \rightarrow \mathbf{Cat}, X \mapsto \mathbf{Cat}[X, \mathbf{Set}]$. It is well known that \mathbf{FSet} satisfies full, faithful, democratic comprehension with Π -types and strong Σ -types (Jacobs 1999). In this context, we have that \mathbf{FLi} has Π -types by Vákár (2017, Theorem 5.2.9). Finally, \mathbf{FLi} indeed has Σ -types and \dashv -types by Vákár (2017, Theorem 5.6.3).

This proves that \mathbf{FLi} is a Σ -bimodel for function types. By Theorem 77, we conclude:

Theorem 78. $\mathbf{FLi} : \mathbf{Set}^{\text{op}} \rightarrow \mathbf{Cat}$ yields a Σ -bimodel for inductive, coinductive, and function types.

Corollary 79. *The categories $\mathbf{Fam}(\mathbf{Li})$ and $\mathbf{Fam}(\mathbf{Li}^{\text{op}})$ are bicartesian closed categories with $\mu\nu$ -polynomials.*

9.5 Fam(Li) and Fam(Li^{op}) are complete and cocomplete

Concrete models provide a significant advantage in terms of the extra properties they can satisfy, which we leverage in our open semantic logical relations. In particular, we have:

Lemma 80. *Fam(Li) and Fam(Li^{op}) are complete and cocomplete.*

Proof. This is a well-known result and, from a fibered perspective, follows from the fact that **FLi** has indexed limits and colimits (and **Set** is cocomplete and complete).

We only need, however, the coproducts and pullbacks that we sketch below.

Coproducts: it is clear that **Fam(Li)** and **Fam(Li^{op})** have coproducts, **Fam(−)** is the cocompletion under coproducts. The coproduct of a (possibly infinite) family $(W_i, w_i)_{i \in L}$ of objects in **Fam(Li)** (respectively **Fam(Li^{op})**) is given by the object $(\bigsqcup_{i \in L} W_i, \langle w_i \rangle_{i \in L})$ in **Fam(Li)** (respectively in **Fam(Li^{op})**), where $\langle w_i \rangle$ denotes the family $\bigsqcup_{i \in L} W_i \rightarrow \mathbf{Li}$ defined by w_i in each component W_i .

Pullbacks: let $(f, f') : (W, w) \rightarrow (Y, y)$ and $(g, g') : (X, x) \rightarrow (Y, y)$ be morphisms of **Fam(Li^{op})**. We consider the pullback $W \times_{(f, g)} X$ of f along g , with projections $p_W : W \times_{(f, g)} X \rightarrow W$ and $p_X : W \times_{(f, g)} X \rightarrow X$. Denoting by s the pushout of (66) in the category **FLi** $(W \times_{(f, g)} X) = \mathbf{Li}^{W \times_{(f, g)} X}$, the pullback of $(f, f') : (W, w) \rightarrow (Y, y)$ and $(g, g') : (X, x) \rightarrow (Y, y)$ in **Fam(Li^{op})** is given by $(W \times_{(f, g)} X, s)$:

$$x \circ p_X \longleftarrow \text{FLi}(p_X)(g') \quad y \circ g \circ p_X = y \circ f \circ p_W \quad \text{FLi}(p_W)(f') \longrightarrow w \circ p_W \tag{66}$$

□

10. Concrete Denotational Semantics for CHAD

In this section, we will establish a concrete denotational semantics for both the source and target languages, and establish CHAD’s specification.

10.1 The concrete model Fam(Set) for the source language

We define a denotational semantics for our source language by interpreting coproducts of Euclidean spaces as families of sets, that is, we interpret our language in **Fam(Set)**. This approach offers technical advantages as it is the natural way to interpret functions between sum types in our setting.

Below, we establish some notation to talk about morphisms, objects, and coproducts in **Fam(Set)**. We start by recalling that the category **Fam(Set)** \simeq **Cat**[2, **Set**] is locally presentable (see Proposition 75). Hence, by Proposition 72, **Fam(Set)** has $\mu\nu$ -polynomials. This proves that **Fam(Set)** is a suitable concrete model for our source language, since **Fam(Set)** \simeq **Cat**[2, **Set**] is cartesian closed.

Proposition 81. *The category Fam(Set) \simeq Cat[2, Set] is complete, cocomplete, cartesian closed and has $\mu\nu$ -polynomials.*

Henceforth, we use the notation $(A_l)_{l \in L} = (L, A^*) \in \mathbf{Fam}(\mathbf{Set})$ to refer to the object of **Fam(Set)** that corresponds to the pair (L, A^*) , where A^* assigns to each $l \in L$ the set A_l . This is a standard way to represent families of sets, where the index set L and the set A_l associated with each index l are explicitly given.

10.1.1 Morphisms between families of sets

Recall that a morphism between families $(A_l)_{l \in L}$ and $(B_i)_{i \in I}$ in **Fam(Set)** is a pair (\underline{f}, f) where $\underline{f} : L \rightarrow I$ is a function and $f = (f_l : A_l \rightarrow B_{f(l)})_{l \in L}$ is family of functions. By abuse of language, we often denote such a morphism (\underline{f}, f) by f , keeping \underline{f} implicit.

10.1.2 Singleton families

For a family $(A_l)_{l \in L} = (L, A^*) \in \text{obj}(\mathbf{Fam}(\mathbf{Set}))$ where $L = \{0\}$ is a singleton, we abuse the notation and write A_0 instead of $(A_l)_{l \in L}$. For example, we use the notation \mathbb{R}^n to denote the singleton family in **Fam(Set)** whose only object is the set \mathbb{R}^n .

In this case, a morphism $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$ in **Fam(Set)** corresponds to a morphism in **Set**. More precisely, the functor **Set** \rightarrow **Fam(Set)** given by $A \mapsto A$ is fully faithful.

10.1.3 Coproducts of families of sets

Let $((A_{(l,i)})_{l \in L_i})_{i \in I} = (L_i, A_i^*)_{i \in I}$ be a (possibly infinite) family of objects of **Fam(Set)**. Recall that the coproduct $\coprod_{i \in I} (L_i, A_i^*)$ in **Fam(Set)** is given by $(\coprod_{i \in I} L_i, \langle A_i^* \rangle_{i \in I})$.

Using the notation established in Section 10.1.2, we see that, for a family of singleton families $(A_i)_{i \in I}$ in **Fam(Set)**, the coproduct $\coprod_{i \in I} A_i$ is the same as the family $(A_i)_{i \in I}$ considered as an object in **Fam(Set)**. Hence, in this context, we often denote by $\coprod_{i \in I} A_i$ the object $(A_i)_{i \in I}$ in **Fam(Set)**.

For instance, consider a family of natural numbers $(n_i)_{i \in I}$, and consider, for each $i \in I$, the object \mathbb{R}^{n_i} of **Fam(Set)**. In this setting, we have that $\coprod_{i \in I} \mathbb{R}^{n_i}$ is the family $(\mathbb{R}^{n_i})_{i \in I}$.

On one hand, it should be noted that, in this setting, a morphism

$$f : \coprod_{i \in I} \mathbb{R}^{n_i} \rightarrow \coprod_{j \in J} \mathbb{R}^{m_j} \tag{67}$$

in **Fam(Set)** is not the same as a function $\coprod_{i \in I} \mathbb{R}^{n_i} \rightarrow \coprod_{j \in J} \mathbb{R}^{m_j}$ in **Set**. More precisely, the functor $\coprod : \mathbf{Fam}(\mathbf{Set}) \rightarrow \mathbf{Set}$ defined by:

$$\left((A_i)_{i \in I} = \coprod_{i \in I} A_i \right) \mapsto \coprod_{i \in I} A_i$$

is not full.

On the other hand, it is worth noting that there is bijection between morphisms of the form (67) in **Fam(Set)** and functions $g : \coprod_{i \in I} \mathbb{R}^{n_i} \rightarrow \coprod_{j \in J} \mathbb{R}^{m_j}$ in **Set** such that, for each $i \in I$, there is $j \in J$ such that $g(\mathbb{R}^{n_i}) \subset \mathbb{R}^{m_j}$.

10.1.4 Products of families of sets

Recall that, given objects $(A_l)_{l \in L}$ and $(B_i)_{i \in I}$ of **Fam(Set)**, the product $(A_l)_{l \in L} \times (B_i)_{i \in I}$ is given by $(A_l \times B_i)_{(l,i) \in L \times I}$.

10.2 The concrete model FVect for the target language

We provide a denotational semantics for our target language by interpreting spaces of (co)tangent vectors as well as derivatives of differentiable functions in terms of families of vector spaces in

Section 10.5.1. To do so, we consider the indexed category $\mathbf{FVect} : \mathbf{Set}^{\text{op}} \rightarrow \mathbf{Cat}$ which associates each set X with \mathbf{Vect}^X .

It should be noted that \mathbf{FVect} is \mathbf{FLi} as considered in Section 9.3 taking $\mathbf{Li} = \mathbf{Vect}$. By Theorem 78:

Corollary 82. $\mathbf{FVect} : \mathbf{Set}^{\text{op}} \rightarrow \mathbf{Cat}$ yields a Σ -bimodel for inductive, coinductive, and function types. Consequently,

$$\Sigma_{\mathbf{Set}}\mathbf{FVect} \cong \mathbf{Fam}(\mathbf{Vect}), \quad \Sigma_{\mathbf{Set}}\mathbf{FVect}^{\text{op}} \cong \mathbf{Fam}(\mathbf{Vect}^{\text{op}}) \tag{68}$$

are bicartesian closed and have $\mu\nu$ -polynomials.

Moreover, by Lemma 80, we have:

Corollary 83. (68) are complete and cocomplete.

We recall some basic aspects of (68) below.

10.2.1 Constant families of vector spaces

We introduce notation for objects in $\mathbf{Fam}(\mathbf{Vect})$ (and $\mathbf{Fam}(\mathbf{Vect}^{\text{op}})$) that correspond to constant families, which is the case for the semantics of our primitive types in the target language. Given a set $N \in \mathbf{Set}$ and a vector space $V \in \mathbf{Vect}$, we denote the corresponding object as (N, \underline{V}) . Here, $\underline{V} : N \rightarrow \mathbf{Vect}$ is the family that is constantly equal to V , meaning that $\underline{V}(s) = V$ for all $s \in N$.

10.2.2 Product of families of vector spaces

Let $(M, m), (N, \nu)$ be objects of $\Sigma_{\mathbf{Set}}\mathbf{FVect} \cong \mathbf{Fam}(\mathbf{Vect})$ (or $\Sigma_{\mathbf{Set}}\mathbf{FVect}^{\text{op}} \cong \mathbf{Fam}(\mathbf{Vect}^{\text{op}})$). By Propositions 17 and 18, we have that

$$(M, m) \times (N, \nu) = (M \times N, (i, j) \mapsto m(i) \times \nu(j)) \tag{69}$$

gives the product of (M, m) and (N, ν) in $\Sigma_{\mathbf{Set}}\mathbf{FVect}$ (and in $\Sigma_{\mathbf{Set}}\mathbf{FVect}^{\text{op}}$). The terminal object in $\Sigma_{\mathbf{Set}}\mathbf{FVect}$ (and in $\Sigma_{\mathbf{Set}}\mathbf{FVect}^{\text{op}}$) is given by $(\mathbb{1}, \mathbb{0})$.

10.2.3 Coproduct of families of vector spaces

Let $(W, w_i)_{i \in L}$ be a family of objects of $\Sigma_{\mathbf{Set}}\mathbf{FVect}$ (or $\Sigma_{\mathbf{Set}}\mathbf{FVect}^{\text{op}}$). We have that (70) gives the coproduct of the family $(W, w_i)_{i \in L}$ in $\Sigma_{\mathbf{Set}}\mathbf{FVect}$ and in $\Sigma_{\mathbf{Set}}\mathbf{FVect}^{\text{op}}$:

$$\left(\coprod_{i \in L} W_i, \langle w_i \rangle_{i \in L} : \coprod_{i \in L} W_i \rightarrow \mathbf{Vect} \right) \tag{70}$$

The initial objects in $\Sigma_{\mathbf{Set}}\mathbf{FVect}$ and in $\Sigma_{\mathbf{Set}}\mathbf{FVect}^{\text{op}}$ are given by $(\emptyset, \mathbb{0})$.

10.2.4 Lists and Streams

Let (71) and (72) be endofunctors on both $\mathbf{Fam}(\mathbf{Vect})$ and $\mathbf{Fam}(\mathbf{Vect}^{\text{op}})$. We can compute the initial algebras and terminal coalgebras of (71) and (72) via colimits and limits of chains (Adámek and Koubek 1979). We get (73) and (74) in both $\mathbf{Fam}(\mathbf{Vect})$ and $\mathbf{Fam}(\mathbf{Vect}^{\text{op}})$:

$$E(X, x) = (\mathbb{1}, 0) \sqcup (X, x) \times (V, \underline{V}) \quad (71) \qquad H(X, x) = (X, x) \times (V, \underline{V}) \quad (72)$$

$$\mu E = \prod_{n=0}^{\infty} (V, \underline{V})^n, \quad (73) \qquad \nu H = \prod_{i=0}^{\infty} (V, \underline{V}) \quad (74)$$

Considering the case where H is an endofunctor on $\mathbf{Fam}(\mathbf{Vect})$, we have that $\nu \hat{H}$ in (76) is the functor constantly equal to the product $\prod_{n=0}^{\infty} V$. When we consider H on $\mathbf{Fam}(\mathbf{Vect}^{\text{op}})$, $\nu \hat{H}$ is the functor constantly equal to $\prod_{i=0}^{\infty} V$.

In the case of the endofunctor E , $\hat{\mu} E$ in (75) is defined by the constant families $\underline{V}^n : V^n \rightarrow \mathbf{Vect}$ in each component V^n of the set $\prod_{i=0}^{\infty} V^n$. This holds true for both $\mathbf{Fam}(\mathbf{Vect})$ and $\mathbf{Fam}(\mathbf{Vect}^{\text{op}})$:

$$\text{List}(V, \underline{V}) = \mu E = \left(\prod_{n=0}^{\infty} V^n, \hat{\mu} E : \prod_{n=0}^{\infty} V^n \rightarrow \mathbf{Vect} \right) \quad (75) \qquad \text{Stream}(V, \underline{V}) = \nu H = \left(\prod_{i=0}^{\infty} V, \nu \hat{H} : \prod_{i=0}^{\infty} V \rightarrow \mathbf{Vect} \right) \quad (76)$$

10.3 Euclidean spaces and coproducts

We introduce the notion of derivatives as it pertains to our work. Our definition aligns with the conventional understanding of derivatives of functions between manifolds, but with added flexibility to accommodate manifolds of varying dimensions. Readers interested in the basics of differentiable manifolds can refer to Lee (2013), Tu (2011).

Let \mathbf{Man} be the category of differentiable manifolds and differentiable maps between them. An *Euclidean space* is an object of \mathbf{Man} that is isomorphic to some differentiable manifold \mathbb{R}^n .

We denote by \mathbf{Diff} the category of Euclidean spaces and differentiable maps between them. In other words, \mathbf{Diff} is the full and replete subcategory of \mathbf{Man} containing the differentiable manifolds \mathbb{R}^k for all $k \in \mathbb{N}$.

Definition 84 (Basic definition of derivatives). *Let $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$ be a morphism in \mathbf{Diff} . We define the morphisms (77) in $\mathbf{Fam}(\mathbf{Vect})$ and (78) in $\mathbf{Fam}(\mathbf{Vect}^{\text{op}})$, where $Df_x := f'(x)$ is the usual Fréchet derivative, and $Df_x^t := f'(x)^t$ is the transpose of $f'(x)$:*

$$\underline{\mathcal{D}}f := (f, Df) : (\mathbb{R}^n, \underline{\mathbb{R}}^n) \rightarrow \prod_{k \in K} (\mathbb{R}^m, \underline{\mathbb{R}}^m) \quad (77)$$

$$\underline{\mathcal{D}}^t f := (f, Df^t) : (\mathbb{R}^n, \underline{\mathbb{R}}^n) \rightarrow \prod_{k \in K} (\mathbb{R}^m, \underline{\mathbb{R}}^m) \quad (78)$$

It follows from the usual properties of derivatives and chain rule that:

Lemma 85 (Derivative of maps between Euclidean spaces). *(77) and (78) uniquely extend to strictly cartesian functors (79) and (80), respectively:*

$$\underline{\mathcal{D}} : \mathbf{Diff} \rightarrow \mathbf{Fam}(\mathbf{Vect}) \quad (79)$$

$$\underline{\mathcal{D}}^t : \mathbf{Diff} \rightarrow \mathbf{Fam}(\mathbf{Vect}^{\text{op}}) \quad (80)$$

While the definitions provided above are presented in the CHAD style, they are essentially the same as the ones used to define derivatives between Euclidean spaces, which are commonly taught in calculus courses.

In order to establish a consistent and rigorous framework for proving the correctness of CHAD for inductive data types, we will extend the definition of derivatives by using *cotupling*. More precisely, from a categorical perspective, this extension will rely on the universal property of the free cocompletion under coproducts.

Definition 86 (Derivative of families). *The universal property of the free cocompletion under coproducts $\mathbf{Fam}(\mathbf{Diff})$ of \mathbf{Diff} induces unique coproduct-preserving functors:*

$$\overline{\mathcal{D}} : \mathbf{Fam}(\mathbf{Diff}) \rightarrow \mathbf{Fam}(\mathbf{Vect}) \tag{81}$$

$$\overline{\mathcal{D}}^t : \mathbf{Fam}(\mathbf{Diff}) \rightarrow \mathbf{Fam}(\mathbf{Vect}^{\text{op}}) \tag{82}$$

that (genuinely) extend the functors (79) and (80), respectively.

Let $\mathbf{Fam}(-)$ be the 2-functor that takes each category to its free cocompletion under coproducts. Denoting by \coprod the respective functors that give the coproduct of families, recall that, by the definition above, (81) and (82) are, respectively, given by the composition (83) and (84):

$$\mathbf{Fam}(\mathbf{Diff}) \xrightarrow{\mathbf{Fam}(\overline{\mathcal{D}})} \mathbf{Fam}(\mathbf{Fam}(\mathbf{Vect})) \xrightarrow{\coprod} \mathbf{Fam}(\mathbf{Vect}) \tag{83}$$

$$\mathbf{Fam}(\mathbf{Diff}) \xrightarrow{\mathbf{Fam}(\overline{\mathcal{D}}^t)} \mathbf{Fam}(\mathbf{Fam}(\mathbf{Vect}^{\text{op}})) \xrightarrow{\coprod} \mathbf{Fam}(\mathbf{Vect}^{\text{op}}) \tag{84}$$

10.4 Euclidean families, differentiable morphisms, derivatives, and diffeomorphisms

We introduce the notion of differentiable morphisms in $\mathbf{Fam}(\mathbf{Set})$, fundamental to establish the specification and correctness of CHAD. To this end, we first define Euclidean families.

Definition 87 (\mathfrak{E} : Euclidean families). *We inductively define the set \mathfrak{E} of Euclidean families by (E1), (E2), and (E3).*

- (E1) For any $k \in \mathbb{N}$, the singleton family with \mathbb{R}^k as a member is an element of \mathfrak{E} .
- (E2) Assuming that A and B are elements of \mathfrak{E} , the product $A \times B$ in $\mathbf{Fam}(\mathbf{Set})$ belongs to \mathfrak{E} .
- (E3) Assuming that $(L_i, A_i^*)_{i \in L}$ is a (possibly infinite) family of objects in \mathfrak{E} , the coproduct

$$\left(\coprod_{i \in L} L_i, \langle A_i^* \rangle_{i \in L} \right) = \coprod_{i \in L} (L_i, A_i^*)$$

in $\mathbf{Fam}(\mathbf{Vect})$ also belongs to \mathfrak{E} .

We denote by:

$$U_e : \mathbf{Fam}(\mathbf{Diff}) \rightarrow \mathbf{Fam}(\mathbf{Set}). \tag{85}$$

the forgetful functor obtained by $U_e := \mathbf{Fam}(U_e)$ where $U_e : \mathbf{Diff} \rightarrow \mathbf{Set}$ denotes the obvious forgetful functor.

Definition 88 (Differentiable morphisms and their derivatives). *A morphism $f : A \rightarrow B$ in $\mathbf{Fam}(\mathbf{Set})$ is differentiable if $A, B \in \mathfrak{E}$ and there is a morphism \mathfrak{f} in $\mathbf{Fam}(\mathbf{Diff})$ such that $U_e(\mathfrak{f}) = f$. In this case, we define*

$$\mathcal{D}f := \overline{\mathcal{D}}\mathfrak{f} \quad \text{and} \quad \mathcal{D}^t f := \overline{\mathcal{D}}^t \mathfrak{f}. \tag{86}$$

We call f a differentiable map, $\mathcal{D}f$ the derivative, and $\mathcal{D}^t f$ the transpose derivative of f .

Definition 89 (Diffeomorphism and diffeomorphic Euclidean families). We say that a morphism f of $\mathbf{Fam}(\mathbf{Set})$ is a diffeomorphism if it is an isomorphism in $\mathbf{Fam}(\mathbf{Vect})$ such that both f and f^{-1} are differentiable.

We say that two objects $(A_l)_{l \in L}$ and $(B_j)_{j \in J}$ of $\mathbf{Fam}(\mathbf{Vect})$ are diffeomorphic if there is a diffeomorphism $(A_l)_{l \in L} \rightarrow (B_j)_{j \in J}$.

It should be noted that the chain rule applies. More precisely:

Lemma 90 (Chain rule). If g and f are composable differentiable morphisms in $\mathbf{Fam}(\mathbf{Set})$, $g \circ f$ is differentiable. Moreover, Eqs. (87) and (88), respectively, hold in $\mathbf{Fam}(\mathbf{Vect})$ and $\mathbf{Fam}(\mathbf{Vect}^{\text{op}})$:

$$\mathfrak{D}g \circ \mathfrak{D}f = \mathfrak{D}(g \circ f) \tag{87} \qquad \mathfrak{D}^t g \circ \mathfrak{D}^t f = \mathfrak{D}^t(g \circ f) \tag{88}$$

We spell out the definition of the derivative of a function between some particular Euclidean families below.

Remark 91 (Explicit derivatives). By Definition 88, (89) in $\mathbf{Fam}(\mathbf{Vect})$ is differentiable if, for each $j \in J$, (90) is differentiable in the usual sense; namely, if (89) is the underlying function of a map $\mathbb{R}^{n_j} \rightarrow \mathbb{R}^{m_{f(j)}}$ in \mathbf{Diff}

$$f = (\underline{f}, f) : \coprod_{j \in J} \mathbb{R}^{n_j} \rightarrow \coprod_{k \in K} \mathbb{R}^{m_k} \tag{89} \qquad f_j : \mathbb{R}^{n_j} \rightarrow \mathbb{R}^{m_{f(j)}} \tag{90}$$

Lemma 93 shows that all differentiable maps can be expressed in the form specified in (91) through the use of canonical diffeomorphisms. More precisely, we show that every Euclidean family is canonically diffeomorphic to something of the form $\coprod_{j \in L} \mathbb{R}^{l_j}$.

Definition 92 (Normal form). For each Euclidean family $A \in \mathfrak{E}$, we inductively define a (possibly infinite) family $\mathcal{N}^{\mathfrak{e}}(A) = (n_j)_{j \in J}$ of natural numbers, and a morphism

$${}^{\mathfrak{e}}n_A : A \rightarrow \coprod_{j \in J} \mathbb{R}^{n_j} \tag{91}$$

in $\mathbf{Fam}(\mathbf{Set})$ by $(\mathcal{N}a)$, $(\mathcal{N}b)$, and $(\mathcal{N}c)$.

$(\mathcal{N}a)$ For each $k \in \mathbb{N}$, $\mathcal{N}^{\mathfrak{e}}(\mathbb{R}^k) := \mathbb{R}^k$ and ${}^{\mathfrak{e}}n_{\mathbb{R}^k} := \text{id}_{\mathbb{R}^k}$.

$(\mathcal{N}b)$ Assuming that $(A, B) \in \mathfrak{E} \times \mathfrak{E}$, $\mathcal{N}^{\mathfrak{e}}(A) = (n_j)_{j \in J}$ and $\mathcal{N}^{\mathfrak{e}}(B) = (m_l)_{l \in L}$, we set

$$\mathcal{N}^{\mathfrak{e}}(A \times B) := (n_j + m_l)_{(j,l) \in J \times L}.$$

We define ${}^{\mathfrak{e}}n_{A \times B}$ by the morphism given by the composition (92), where the unlabeled arrow is the canonical isomorphism induced by the universal property of the product and the distributive property of \mathbf{Set}

$$A \times B \xrightarrow{{}^{\mathfrak{e}}n_A \times {}^{\mathfrak{e}}n_B} \coprod_{j \in J} \mathbb{R}^{n_j} \times \coprod_{l \in L} \mathbb{R}^{m_l} \longrightarrow \coprod_{(j,l) \in J \times L} \mathbb{R}^{n_j + m_l} \tag{92}$$

$(\mathcal{N}c)$ Assuming that $(L_j, A_j^*)_{j \in J}$ is a family of objects in \mathfrak{E} such that $\mathcal{N}^{\mathfrak{e}}((L_j, A_j^*)) = (m_{(j,l)})_{l \in L_j}$, we set

$${}^{\mathfrak{e}}\left(\coprod_{j \in J} A_j\right) := (m_t)_{t \in I},$$

where $l := \bigcup_{j \in J} \{j\} \times L_j$. Finally, we define $\epsilon_{\coprod_{j \in J} A_j}$ by the composition (93) where the unlabeled arrow is the canonical isomorphism induced by the universal property of coproducts:

$$\coprod_{j \in J} A_j \xrightarrow{\coprod_{j \in J} \epsilon_{n_{A_j}}} \coprod_{j \in J} \left(\coprod_{l \in L_j} \mathbb{R}^{m_{(j,l)}} \right) \longrightarrow \coprod_{t \in l} \mathbb{R}^{m_t} \tag{93}$$

It is simple to verify by induction that:

Lemma 93 (Canonical form of Euclidean families). *For every object $A \in \mathcal{E}$, ϵ_{n_A} is a diffeomorphism.*

By utilizing these normal forms, we are able to establish a valuable characterization of differentiable maps (Lemma 94). This characterization is then leveraged in our logical relations argument, which is detailed in Sections 12 and 13.

Lemma 94. *Let $f : W \rightarrow X$ be a morphism in $\mathbf{Fam}(\mathbf{Set})$ and (g, h) a morphism in $\mathbf{Fam}(\mathbf{Vect}) \times \mathbf{Fam}(\mathbf{Vect}^{\text{op}})$. Assuming that $W \in \mathcal{E}$, we have that*

$$\begin{aligned} & f \text{ is differentiable and } (g, h) = (\mathcal{D}f, \mathcal{D}^t f) \\ & \text{if, and only if,} \\ & f \circ \gamma \text{ is differentiable, } g \circ \mathcal{D}\gamma = \mathcal{D}(f \circ \gamma), \text{ and } h \circ \mathcal{D}^t \gamma = \mathcal{D}^t(f \circ \gamma) \end{aligned}$$

for any differentiable map $\gamma : \mathbb{R}^n \rightarrow W$ in $\mathbf{Fam}(\mathbf{Set})$ (where n is any natural number).

Proof. It should be noted that one direction follows from chain rule; namely, if f is differentiable and $(g, h) = (\mathcal{D}f, \mathcal{D}^t f)$, then $f \circ \alpha$ is differentiable, $g \circ \mathcal{D}\alpha = \mathcal{D}(f \circ \alpha)$, and $h \circ \mathcal{D}^t \alpha = \mathcal{D}^t(f \circ \alpha)$.

Reciprocally, we assume that f and (g, h) are such that $f \circ \gamma$ is differentiable, $g \circ \mathcal{D}\gamma = \mathcal{D}(f \circ \gamma)$, and $h \circ \mathcal{D}^t \gamma = \mathcal{D}^t(f \circ \gamma)$ for any differentiable map $\gamma : \mathbb{R}^n \rightarrow W$ in $\mathbf{Fam}(\mathbf{Set})$.

Since $W \in \mathcal{E}$, we conclude that so is X since, by hypothesis, we can conclude that there is at least a morphism $W \rightarrow X$ that is differentiable.

By Lemma 93, we have canonical diffeomorphism:

$$\epsilon_{n_W} : W \rightarrow \coprod_{j \in J} \mathbb{R}^{n_j}, \quad \epsilon_{n_X} : X \rightarrow \coprod_{l \in L} \mathbb{R}^{m_l}$$

where $(n_j)_{j \in J} = \mathcal{N}^\epsilon(W)$ and $(m_l)_{l \in L} = \mathcal{N}^\epsilon(X)$ as defined in Definition 92.

For each $j \in J$, we define $\gamma_j := \epsilon_{n_W} \circ \iota_{\mathbb{R}^{n_j}}$ where

$$\iota_{\mathbb{R}^{n_j}} : \mathbb{R}^{n_j} \rightarrow \coprod_{j \in J} \mathbb{R}^{n_j}$$

is the coproduct coprojection in $\mathbf{Fam}(\mathbf{Set})$. By hypothesis, for all $j \in J$,

$$f \circ \epsilon_{n_W} \circ \iota_{\mathbb{R}^{n_j}} = f \circ \gamma_j$$

is differentiable and, hence, $\epsilon_{n_X} \circ f \circ \epsilon_{n_W} \circ \iota_{\mathbb{R}^{n_j}} = \epsilon_{n_X} \circ f \circ \gamma_j$ is differentiable by the chain rule. This shows that

$$\epsilon_{n_X} \circ f \circ \epsilon_{n_W} : \coprod_{j \in J} \mathbb{R}^{n_j} \rightarrow X \tag{94}$$

is componentwise differentiable, that is to say, (94) is such that

$$({}^c n_X \circ f \circ {}^c n_W)_{j \in J} : \mathbb{R}^{n_j} \rightarrow \mathbb{R}^{m_{n_X \circ f \circ {}^c n_W(j)}}$$

is differentiable for all $j \in J$. By Remark 91, we conclude that ${}^c n_X \circ f \circ {}^c n_W$ is differentiable. Since ${}^c n_X$ and ${}^c n_W$ are diffeomorphisms, this proves that f is differentiable by the chain rule.

Analogously, by using the morphisms γ_j defined above, we conclude that $(\mathcal{D}({}^c n_X) \circ g \circ \mathcal{D}({}^c n_W), \mathcal{D}^t({}^c n_X) \circ h \circ \mathcal{D}^t({}^c n_W)) = (\mathcal{D}({}^c n_X \circ f \circ {}^c n_W), \mathcal{D}^t({}^c n_X \circ f \circ {}^c n_W))$. Therefore, since ${}^c n_W$ and ${}^c n_X$ are diffeomorphisms, $(g, h) = (\mathcal{D}f, \mathcal{D}^t f)$ by the chain rule. \square

10.5 Semantic functors

We establish the concrete denotational semantics of our languages as suitable structure-preserving functors induced by the respective universal properties.

10.5.1 The concrete denotational model for the source language

Recall that **Fam(Set)** is cartesian closed and has $\mu\nu$ -polynomials (Proposition 81). By the universal property of the source language **Syn** established in Corollary 15, we can define the semantic functor from **Syn** to **Fam(Set)**:

Corollary 95 (Concrete semantics of the source language). *We fix the concrete semantics of the ground types and primitive operations of **Syn** by defining*

- (s-a) for each n -dimensional array $\mathbf{real}^n \in \mathbf{Syn}$, $\llbracket \mathbf{real}^n \rrbracket \stackrel{\text{def}}{=} \mathbb{R}^n \in \text{obj}(\mathbf{Fam}(\mathbf{Set}))$ in which \mathbb{R}^n is the singleton family with \mathbb{R}^n as unique member,
- (s-b) for each primitive $\text{op} \in \text{Op}_{n_1, \dots, n_k}^m$, $\llbracket \text{op} \rrbracket : \mathbb{R}^{n_1} \times \dots \times \mathbb{R}^{n_k} \rightarrow \mathbb{R}^m$ is the map in **Fam(Set)** corresponding to the function that op intends to implement.

By Corollary 15, we obtain a unique functor:

$$\llbracket - \rrbracket : \mathbf{Syn} \rightarrow \mathbf{Set}$$

that extends these definitions to give a concrete denotational semantics for the entire source language such that $\llbracket - \rrbracket$ is a strictly bicartesian closed functor that (strictly) preserves $\mu\nu$ -polynomials.

10.5.2 The concrete denotational model for the target language

We establish the concrete denotational semantics of our target language. Recall that **FVect** is a Σ -bimodel for tuples, function types, sum types, and inductive and coinductive types by Corollary 82.

We define the functors (95) and (96) induced by the universal property of (**CSyn**, **LSyn**) established in Corollary 68.

Henceforth, we make use of the terminology and notation established in Sections 10.4 and 91.

Corollary 96 (Concrete semantics of the target language). *Let $\mathbf{FVect} : \mathbf{Set}^{\text{op}} \rightarrow \mathbf{Cat}$ be the Σ -bimodel for inductive, coinductive, and function types $\mathbf{FVect} : \mathbf{Set}^{\text{op}} \rightarrow \mathbf{Cat}$ established in Section 10.2 (see Corollary 82). We establish the following assignment:*

- (t-a) for each n -dimensional array $\mathbf{real}^n \in \mathbf{Syn}$, $\overline{\Sigma \llbracket \mathbf{real}^n \rrbracket} = \overline{\Sigma \llbracket \mathbf{real}^n \rrbracket} \stackrel{\text{def}}{=} \llbracket \mathbf{real}^n \rrbracket \mathbb{R}^n \in \mathbf{Set}$;
- (t-b) for each n -dimensional array $\mathbf{real}^n \in \mathbf{Syn}$,

$$\Sigma \llbracket \mathbf{real}^n \rrbracket = \overline{\Sigma \llbracket \mathbf{real}^n \rrbracket} \stackrel{\text{def}}{=} L_{\mathbf{real}^n} \in \mathbf{FVect}(\mathbb{R}^n)$$

in which $L_{\mathbf{real}^n} = \underline{\mathbb{R}^n} : \mathbb{R}^n \rightarrow \mathbf{Vect}$;

(t-c) for each primitive $\text{op} \in \text{Op}_{n_1, \dots, n_k}^m$:

- (t-i) $\overline{\Sigma[\text{op}]} = \llbracket \text{op} \rrbracket : \mathbb{R}^{n_1} \times \dots \times \mathbb{R}^{n_k} \rightarrow \mathbb{R}^m$ is the map in **Set** corresponding to the function that op intends to implement;
- (t-ii) $f_{\text{op}} = \llbracket \text{Dop} \rrbracket \in \mathbf{FVect}(\mathbb{R}^{n_1} \times \dots \times \mathbb{R}^{n_k})(\mathbb{R}^{n_1} \times \dots \times \mathbb{R}^{n_k}, \mathbb{R}^m)$ is the family of linear transformations that Dop intends to implement;
- (t-iii) $f_{\text{op}}^t = \llbracket (\text{Dop})^t \rrbracket \in \mathbf{FVect}(\mathbb{R}^{n_1} \times \dots \times \mathbb{R}^{n_k})(\mathbb{R}^m, \mathbb{R}^{n_1} \times \dots \times \mathbb{R}^{n_k})$ is the family of linear transformations that Dop^t intends to implement.

By Corollary 68, we obtain canonical functors:

$$\Sigma[-] : \Sigma_{\text{CSyn}} \mathbf{LSyn} \rightarrow \Sigma_{\text{Set}} \mathbf{FVect} \cong \mathbf{Fam}(\mathbf{Vect}) \tag{95}$$

$$\Sigma^t[-] : \Sigma_{\text{CSyn}} \mathbf{LSyn}^{\text{op}} \rightarrow \Sigma_{\text{Set}} \mathbf{FVect}^{\text{op}} \cong \mathbf{Fam}(\mathbf{Vect}^{\text{op}}) \tag{96}$$

that extend (t-a), (t-b), and (t-c) to give a concrete denotational semantics for the entire target language of the forward AD and the reverse AD, respectively, such that $\Sigma[-]$ and $\Sigma^t[-]$ are bicartesian closed functors that preserve $\mu\nu$ -polynomials.

10.6 Semantic assumptions and specification of CHAD

Although our work applies to more general contexts, we assume that every primitive operation in the source language intends to implement a differentiable function. We claim that, whenever we have an AD correct macro in this setting, this can be applied to further general cases. For the case of dual-numbers AD, we refer to the revised version of Lucatelli Nunes and Vákár (2022a) for comments on general contexts involving nondifferentiable functions.

More precisely, for any primitive operation $\text{op} \in \text{Op}_{n_1, \dots, n_k}^m$ of the source language, we assume that

$$\llbracket \text{op} \rrbracket : \prod_{i=1}^k \mathbb{R}^{n_i} \rightarrow \mathbb{R}^m$$

is differentiable. Moreover, we assume that (97) and (98) hold

$$\Sigma[\overrightarrow{\mathcal{D}}(\text{op})] = \mathfrak{D}\llbracket \text{op} \rrbracket, \tag{97} \quad \Sigma^t[\overleftarrow{\mathcal{D}}(\text{op})] = \mathfrak{D}^t\llbracket \text{op} \rrbracket. \tag{98}$$

It should be noted that (98) and (97) hold as long as $\Sigma[\overrightarrow{\mathcal{D}}(\text{op})] = (\llbracket \text{op} \rrbracket, \llbracket \text{Dop} \rrbracket) = (\llbracket \text{op} \rrbracket, f_{\text{op}})$ and $\Sigma^t[\overleftarrow{\mathcal{D}}(\text{op})] = (\llbracket \text{op} \rrbracket, \llbracket \text{Dop}^t \rrbracket) = (\llbracket \text{op} \rrbracket, f_{\text{op}}^t)$. In other words, (98) and (97) hold as long as Dop and Dop^t implement the family of linear transformations corresponding to the respective derivatives of $\llbracket \text{op} \rrbracket$, as explained in Section 7

10.6.1 Specification

We can inductively define what we mean by *data types* in the source language. These are those types constructed out of ground types, tuples, variant types, and inductive types.

We show in Section 13 that the semantics for the inductive data types are rather simple, as they are Euclidean families, that is to say, elements of \mathfrak{E} . This shows, by Lemma 93, that the semantics of any data type is isomorphic (actually, canonically diffeomorphic) to a (possibly infinite) coproduct of $\prod_{j \in J} \mathbb{R}^{n_j}$.

In Section 13, we prove the full correctness theorem of CHAD for data types. More precisely, given any well-typed program $x_1 : \tau \vdash t : \sigma$ in the source language, where τ, σ are data types, we have that:

- (C1) $\llbracket \tau \rrbracket$ and $\llbracket \sigma \rrbracket$ are Euclidean families;
- (C2) $\llbracket t \rrbracket$ is differentiable;
- (C3) $\xrightarrow{\Sigma} \llbracket \overrightarrow{\mathcal{D}}(t) \rrbracket = \mathfrak{D} \llbracket t \rrbracket$ and $\overset{t}{\Sigma} \llbracket \overleftarrow{\mathcal{D}}(t) \rrbracket = \mathfrak{D}^t \llbracket t \rrbracket$.

11. Scoring

Our approach to categorical semantics for logical relations emphasizes principled constructions of concrete categories from elementary ones, guided by the properties we seek to prove in each setting, for example, Lucatelli Nunes and Vákár (2022b, Section 4). In this section, we introduce the basic categorical framework for our open semantic logical relations proof; namely, we study the *scone*, also called *Artin gluing*.

Recall that, given a functor $G : \mathcal{C} \rightarrow \mathcal{D}$, the *scone* of G is the comma category $\mathcal{D} \downarrow G$ of the identity along G . Explicitly, the scone’s objects are triples $(C_0 \in \mathcal{D}, C_1 \in \mathcal{C}, f : C_0 \rightarrow G(C_1))$ in which f is a morphism of \mathcal{D} . Its morphisms $(C_0, C_1, f) \rightarrow (C'_0, C'_1, f')$ are pairs $(h_0 : C_0 \rightarrow C'_0, h_1 : C_1 \rightarrow C'_1)$ such that (99) commutes in \mathcal{D} :

$$\begin{array}{ccc}
 C_0 & \xrightarrow{h_0} & C'_0 \\
 f \downarrow & & \downarrow f' \\
 G(C_1) & \xrightarrow{G(h_1)} & G(C'_1)
 \end{array} \tag{99}$$

The scone $\mathcal{D} \downarrow G$ inherits much of the structure of $\mathcal{D} \times \mathcal{C}$. For that reason, under suitable conditions, scoring can be seen as a principled way of building a suitable categorical model from a previously given categorical model $\mathcal{D} \times \mathcal{C}$, providing an appropriate semantics for our problem. This is, indeed, the fundamental aspect that underlies our logical relations argument in Section 12 and also in Vákár and Smeding (2022) and Lucatelli Nunes and Vákár (2022a,b).

In this section, we present our comonadic–monadic approach to study the properties of $\mathcal{D} \downarrow G$; it is consisting of studying $\mathcal{D} \downarrow G$ via its comonadicity and monadicity over $\mathcal{D} \times \mathcal{C}$. This approach allows us to establish conditions under which $\mathcal{D} \downarrow G$ has $\mu\nu$ -polynomials. The key contribution of this section is twofold: (1) our approach provides a systematic and principled way to understand the nice properties of $\mathcal{D} \downarrow G$ under suitable conditions; and (2) the conditions we establish for the existence of $\mu\nu$ -polynomials are particularly useful for building categorical models for logical relations arguments.

Specifically, our approach shows that the forgetful functor:

$$L : \mathcal{D} \downarrow G \rightarrow \mathcal{D} \times \mathcal{C} \tag{100}$$

is comonadic and, in our case, monadic, and that the properties of $\mathcal{D} \downarrow G$ can be seen as consequences of this fact.

To lay the groundwork for our approach, we begin by recalling Beck’s Monadicity Theorem, since Theorem 97 holds a fundamental place in our approach. The original statement of this theorem involves split (co)equalizers; see, for instance, Barr and Wells (2005, Theorem3.14) or Dubuc (1970, TheoremII.2.1) for the enriched case. However, for our purposes, we will make use of a slightly modified version, namely *a left adjoint functor is comonadic if and only if it creates absolute limits*. This version can be found, for instance, in Lucatelli Nunes (2021, p. 550).

Theorem 97. *If \mathcal{D} has binary products, then (100) is comonadic.*

Proof. By the universal property of comma categories, a diagram $D : \mathcal{S} \rightarrow \mathcal{D} \downarrow G$ corresponds biunivocally with triples:

$$(D_0 : \mathcal{S} \rightarrow \mathcal{D}, D_1 : \mathcal{S} \rightarrow \mathcal{C}, \vartheta : D_0 \rightarrow GD_1) \tag{101}$$

in which D_0, D_1 are diagrams and ϑ is a natural transformation. In this setting, it is clear that, assuming that $\lim D_0$ exists, if $\lim D_1$ exists and is preserved by G , we have that

$$\left(\lim D_0, \lim D_1, \lim D_0 \xrightarrow{d} \lim (G \circ D_1) \xrightarrow{\cong} G(\lim D_1) \right), \tag{102}$$

is the limit of D in $\mathcal{D} \downarrow G$, in which d is the morphism induced by the natural transformation ϑ .

Now, given a diagram $D : \mathcal{S} \rightarrow \mathcal{D} \downarrow G$ such that $L \circ D = (D_0, D_1) : \mathcal{S} \rightarrow \mathcal{D} \times \mathcal{C}$ has an absolute limit, we get that $\lim D_0$ and $\lim D_1$ exist and are preserved by any functor. Hence, by the observed above, in this case, the limit of D exists and is given by (102). Thus, it is preserved by L . Since (100) is conservative, this completes the proof that (100) creates absolute limits.

Finally, since (100) has a right adjoint defined by:

$$(Y, X) \mapsto (Y \times G(X), X, \pi_2 : Y \times G(X) \rightarrow G(X)),$$

the proof that (100) is comonadic is complete by Beck’s Monadicity Theorem. □

Remark 98. If \mathcal{C} has a terminal object and \mathcal{D} has binary products as above, (100) is comonadic and; furthermore, the comonad induced by it is the free comonad over the endofunctor on $\mathcal{D} \times \mathcal{C}$ defined by $(Y, X) \mapsto (G(X), \mathbb{1})$.

Corollary 99. Assume that \mathcal{C} has binary coproducts and \mathcal{D} has binary products. We have that (100) is comonadic and monadic provided that G has a left adjoint F .

Proof. Firstly, of course, by Theorem 97, we have that (100) is comonadic. Secondly, by the dual of Theorem 97, we have that the forgetful functor $F \downarrow \mathcal{C} \rightarrow \mathcal{C} \times \mathcal{D}$ is monadic. Hence, since

$$\begin{array}{ccc} \mathcal{D} \downarrow G & \xleftarrow{\quad L \quad} & \mathcal{D} \times \mathcal{C} \\ & \cong \searrow & \nearrow \\ & & F \downarrow \mathcal{C} \end{array}$$

commutes, we get that L is monadic as well. □

Indeed, in our case, all the properties of the scone we are interested in follow from the comonadicity and monadicity of (100), that is to say, Corollary 99.⁸

11.1 Bicartesian structure of the scone

The bicartesian closed structure of the scone $\mathcal{D} \downarrow G$ follows from the well-known result about monadic functors and creation of limits. Namely:

Proposition 100. Monadic functors create all limits. Dually, comonadic functors create all colimits.

Proof. See, for instance, MacDonald and Sobral (2004, Section 1.4). □

As a corollary, then, we have the following explicit constructions.

Corollary 101. Assuming that \mathcal{C} and \mathcal{D} have finite products and finite coproducts, if $G : \mathcal{C} \rightarrow \mathcal{D}$ has a left adjoint, then $L : \mathcal{D} \downarrow G \rightarrow \mathcal{D} \times \mathcal{C}$ creates limits and colimits. In particular, $\mathcal{D} \downarrow G$ is bicartesian and, in the case $\mathcal{D} \times \mathcal{C}$ is a distributive category, so is $\mathcal{D} \downarrow G$.

Proof. Given a diagram $D : \mathcal{S} \rightarrow \mathcal{D} \downarrow G$, we have that it is uniquely determined by a triple $(D_0 : \mathcal{S} \rightarrow \mathcal{D}, D_1 : \mathcal{S} \rightarrow \mathcal{C}, \delta : D_0 \rightarrow GD_1)$ like in (101). In this case, we have that:

- (1) In the proof of Theorem 97, we implicitly addressed the problem of creation of limits that are preserved by G . Since G has a left adjoint, it preserves all the limits and, hence, all the limits are created like (103).
 More precisely, assuming that $L \circ D = (D_0, D_1) : \mathcal{S} \rightarrow \mathcal{D} \times \mathcal{C}$ has a limit, we get that both $\lim D_0$ and $\lim D_1$ exist, since the projections $\mathcal{D} \times \mathcal{C} \rightarrow \mathcal{D}$ and $\mathcal{D} \times \mathcal{C} \rightarrow \mathcal{C}$ have left adjoints (because \mathcal{C} and \mathcal{D} have initial objects).
 Since G has a left adjoint, it preserves the limit of D_1 . Hence, the limit of D is given by:

$$\left(\lim D_0, \lim D_1, \lim D_0, \xrightarrow{d} \lim (G \circ D_1) \xrightarrow{\cong} G(\lim D_1) \right), \tag{103}$$

like in (102), in which d is the morphism induced by δ and $\lim (G \circ D_1) \cong G(\lim D_1)$ comes from the fact that G preserves limits.

- (2) Assuming that $L \circ D = (D_0, D_1) : \mathcal{S} \rightarrow \mathcal{D} \times \mathcal{C}$ has a limit, we get that both $\text{colim } D_0$ and $\text{colim } D_1$ exist. In this case, the colimit of D is given by:

$$\left(\text{colim } D_0, \text{colim } D_1, \text{colim } D_0 \xrightarrow{d} \text{colim } (G \circ D_1) \rightarrow G(\text{colim } D_1) \right), \tag{104}$$

in which $\text{colim } (G \circ D_1) \rightarrow G(\text{colim } D_1)$ is the induced comparison. □

Remark 102. It will be particularly important for our correctness proof in Section 13 that $\mathcal{D} \downarrow G$ has infinite coproducts whenever \mathcal{C} and \mathcal{D} have finite products and infinite coproducts. This is a consequence of the fact stated above.

11.2 Monadic-comonadic functors and the cartesian closedness of the scone

Under the conditions of our proof, the scone $\mathcal{D} \downarrow G$ is cartesian closed. In our case, we can see as a consequence of the well-known result below.

Proposition 103. *If a category is monadic-comonadic over a finitely complete cartesian closed category, then it is finitely complete cartesian closed as well.*

More precisely, if \mathcal{D} is finitely complete and $G : \mathcal{C} \rightarrow \mathcal{D}$ is monadic and comonadic, then G reflects exponentiable objects.

Proof. See, for instance, a slightly more general version in Lucatelli Nunes (2017, Theorem 1.8.2). Indeed, assuming that $G : \mathcal{C} \rightarrow \mathcal{D}$ is monadic and comonadic and that \mathcal{D} is finitely complete, we get that \mathcal{C} is finitely complete as well and, moreover, G preserves them (since monadic functors create limits).

Denoting the right adjoint of G by H , given an object $W \in \mathcal{C}$, we have an isomorphism:

$$\begin{array}{ccc} \mathcal{C} & \xrightarrow{(W \times -)} & \mathcal{C} \\ G \downarrow & \cong & \downarrow G \\ \mathcal{D} & \xrightarrow{(G(W) \times -)} & \mathcal{D} \end{array} \tag{105}$$

If $G(W)$ is exponentiable, we know that $(G(W) \times G(-)) \dashv H(G(W) \Rightarrow -)$. Since \mathcal{C} has equalizers and G is comonadic, we get that $(W \times -)$ has a right adjoint by Dubuc’s adjoint triangle theorem.⁹ That is to say, W is exponentiable. \square

Explicitly, we get:

Corollary 104. *Let \mathcal{C} and \mathcal{D} be finitely complete cartesian closed categories. If $G : \mathcal{C} \rightarrow \mathcal{D}$ has a left adjoint, we get that $\mathcal{D} \downarrow G$ is finitely complete cartesian closed. More precisely, the exponential in $\mathcal{D} \downarrow G$ is given by (106) where we write $f \Rightarrow f'$ for the Pullback (107):*

$$(C_0, C_1, f) \Rightarrow (D_0, D_1, f') = (P, C_1 \Rightarrow D_1, f \Rightarrow f') \tag{106}$$

$$\begin{array}{ccc}
 P & \xrightarrow{\quad\quad\quad} & C_0 \Rightarrow D_0 \\
 \downarrow f \Rightarrow f' & & \downarrow C_0 \Rightarrow f' \\
 G(C_1 \Rightarrow D_1) & \xrightarrow{\quad\quad\quad} & G(C_1) \Rightarrow G(D_1) \xrightarrow{f \Rightarrow G(D_1)} C_0 \Rightarrow G(D_1)
 \end{array} \tag{107}$$

11.3 Monadic functors create terminal coalgebras of compatible endofunctors

Recall the definition of preservation, reflection, and creation of initial algebras and terminal coalgebras; see Definitions 11 and 13. We prove and establish the result that says that monadic functors create initial algebras, while, dually, comonadic functors create terminal coalgebras.

We first establish the fact that left adjoint functors preserve initial algebras and, dually, right adjoint functors preserve terminal coalgebras. In order to do so, we start by observing that:

Lemma 105. *Let*

$$\begin{array}{ccc}
 & F & \\
 C & \xleftarrow{\quad\quad\quad} & D \\
 & \perp(\varepsilon, \eta) & \\
 & G &
 \end{array}$$

be an adjunction. Assume that $\gamma : E \circ F \cong F \circ E'$ is a natural isomorphism in which E and E' are endofunctors. In this case, we have an induced adjunction:

$$\begin{array}{ccc}
 & \check{F}_\gamma & \\
 E\text{-Alg} & \xleftarrow{\quad\quad\quad} & E'\text{-Alg} \\
 & \perp(\hat{\varepsilon}, \hat{\eta}) & \\
 & \hat{G}_\gamma &
 \end{array} \tag{108}$$

in which \check{F}_γ is defined as in Definition 11 and \hat{G}_γ is defined as follows:

$$\begin{aligned}
 \hat{G}_\gamma : E\text{-Alg} &\rightarrow E'\text{-Alg} \\
 (Y, \xi) &\mapsto \left(G(Y), G(\xi) \circ GE(\varepsilon_Y) \circ G(\gamma_{G(Y)}^{-1}) \circ \eta_{E'G(Y)} \right) \\
 f &\mapsto G(f).
 \end{aligned}$$

Proof. In fact, the counit and unit, $\hat{\varepsilon}, \hat{\eta}$, are defined pointwise by the original counit and unit. That is to say, $\hat{\varepsilon}_{(Y, \xi)} = \varepsilon_Y$ and $\hat{\eta}_{(W, \xi)} = \eta_W$. \square

Remark 106 (Doctrinal adjunction). The right adjoint \hat{G}_γ does not come out of the blue. The association $(F, \gamma) \mapsto \check{F}\gamma$ in Lemma 9 is part of a 2-functor, with the domain being the 2-category of endomorphisms in **Cat**, lax natural transformations and modifications, and the codomain being **Cat**. By the doctrinal adjunction,¹⁰ we know that whenever (F, γ) is pseudo-natural (i.e., γ is invertible) and F has a right adjoint in **Cat**, the pair (F, γ) has a right adjoint $(G, (GE\varepsilon) \cdot (G\gamma_G^{-1}) \cdot (\eta E'G))$ in the 2-category of endofunctors. Therefore, since 2-functors preserve adjunctions, we obtain that $\check{F}\gamma$ has a right adjoint given by $\check{G}_{(GE\varepsilon) \cdot (G\gamma_G^{-1}) \cdot (\eta E'G)}$, denoted by \hat{G}_γ , whenever γ is invertible and F has a right adjoint.

The dual of Lemma 105 is given by:

Lemma 107. *Let*

$$\begin{array}{ccc} & F & \\ & \curvearrowright & \\ \mathcal{C} & \perp(\varepsilon, \eta) & \mathcal{D} \\ & \curvearrowleft & \\ & G & \end{array}$$

be an adjunction. Assume that $\beta : G \circ E \cong E' \circ G$ is a natural isomorphism in which E and E' are endofunctors. In this case, we have an induced adjunction:

$$\begin{array}{ccc} & \hat{F}^\beta & \\ & \curvearrowright & \\ E\text{-CoAlg} & \perp(\hat{\varepsilon}, \hat{\eta}) & E'\text{-CoAlg} \\ & \curvearrowleft & \\ & \tilde{G}^\beta & \end{array} \tag{109}$$

in which \tilde{G}^β is defined as in Definition 13 and \hat{F}^β is defined as follows:

$$\begin{aligned} \hat{F} : E'\text{-CoAlg} &\rightarrow E\text{-CoAlg} \\ (W, \zeta) &\mapsto (W, \varepsilon_{EF(W)} \circ F(\beta_{F(W)}^{-1}) \circ FE'(\eta_W) \circ F(\zeta)) \\ g &\mapsto F(g). \end{aligned}$$

As an immediate consequence, we have that:

Theorem 108. *Right adjoint functors preserve terminal coalgebras. Dually, left adjoints preserve initial algebras.*

Proof. Let $G : \mathcal{C} \rightarrow \mathcal{D}$ be a functor and $\beta : G \circ E \cong E' \circ G$ a natural isomorphism in which E, E' are endofunctors. If $F \dashv G$, we get that $\tilde{G}^\beta : E\text{-CoAlg} \rightarrow E'\text{-CoAlg}$ (as defined in Definition 13) has a left adjoint by Lemma 107. Therefore, \tilde{G}^β preserves limits and, in particular, terminal objects. This completes the proof that G preserves terminal coalgebras (see Definition 13). \square

Finally, we can state the result about monadic functors; namely:

Theorem 109. *Monadic functors create terminal coalgebras. Dually, comonadic functors create initial algebras.*

Proof. Let $G : \mathcal{C} \rightarrow \mathcal{D}$ be a monadic functor. Assume that $\beta : G \circ E \cong E' \circ G$ is a natural isomorphism in which E, E' are endofunctors.

We have that $\tilde{G}^\beta : E\text{-CoAlg} \rightarrow E'\text{-CoAlg}$ (as defined in Definition 13) has a left adjoint by Lemma 107. Moreover, we have the commutative diagram:

$$\begin{array}{ccc}
 E\text{-CoAlg} & \xrightarrow{\tilde{G}^\beta} & E'\text{-CoAlg} \\
 \downarrow & & \downarrow \\
 \mathcal{C} & \xrightarrow{G} & \mathcal{D}
 \end{array} \tag{110}$$

in which the vertical arrows are the forgetful functors.

Since we know that all the functors in (110) but \tilde{G}^β create absolute colimits, we conclude that \tilde{G}^β creates absolute colimits as well. Therefore, \tilde{G}^β is monadic and, thus, it creates all limits. In particular, \tilde{G}^β creates terminal objects. This completes the proof that G creates terminal coalgebras (see Definition 13). \square

11.4 Monadic-comonadic functors create $\mu\nu$ -polynomials

We establish that monadic-comonadic functors create $\mu\nu$ -polynomials below, a crucial result for our approach to the study of $\mu\nu$ -polynomials in the scene.

Corollary 110. *Monadic-comonadic functors create $\mu\nu$ -polynomials. More precisely, if $G : \mathcal{A} \rightarrow \mathcal{B}$ is monadic-comonadic and \mathcal{B} has $\mu\nu$ -polynomials, then*

- (1) G creates products and coproducts;
- (2) \mathcal{A} has $\mu\nu$ -polynomials;
- (3) for each $\mu\nu$ -polynomial endofunctor E on \mathcal{A} , there is a $\mu\nu$ -polynomial endofunctor \bar{E} on \mathcal{B} such that $G \circ E \cong \bar{E} \circ G$ (and G creates the initial algebra and the terminal coalgebra of E).

Proof. Let $G : \mathcal{A} \rightarrow \mathcal{B}$ be a monadic-comonadic functor in which \mathcal{B} has $\mu\nu$ -polynomials. We inductively define the set $\overline{\times G}$ as follows:

- $(\overline{\times G}1)$ the identity functor $\mathbb{1} \rightarrow \mathbb{1}$ belongs to $\overline{\times G}$;
- $(\overline{\times G}2)$ $G : \mathcal{A} \rightarrow \mathcal{B}$ belongs to $\overline{\times G}$;
- $(\overline{\times G}3)$ if $G' : \mathcal{A}' \rightarrow \mathcal{B}'$ and $G'' : \mathcal{A}'' \rightarrow \mathcal{B}''$ belong $\overline{\times G}$, then so does the product $G' \times G'' : \mathcal{A}' \times \mathcal{A}'' \rightarrow \mathcal{B}' \times \mathcal{B}''$.

We have bijections (111) and (112) inductively defined by ((bi)), ((bii)), and ((biii)):

$$\text{dom} : \overline{\times G} \rightarrow \text{obj}(\mu\nu\text{Poly}_{\mathcal{A}}) \tag{111} \qquad \text{codom} : \overline{\times G} \rightarrow \text{obj}(\mu\nu\text{Poly}_{\mathcal{B}}) \tag{112}$$

- (bi) $\text{codom}(\mathbb{1} \rightarrow \mathbb{1}) = \text{dom}(\mathbb{1} \rightarrow \mathbb{1}) = \mathbb{1}$;
- (bii) $\text{dom}(G) = \mathcal{A}$ and $\text{codom}(G) = \mathcal{B}$;
- (biii) $\text{dom}(G' \times G'') = \text{dom}(G') \times \text{dom}(G'')$ and $\text{codom}(G' \times G'') = \text{codom}(G') \times \text{codom}(G'')$.

In other words, the function $\text{dom} : \overline{\times G} \rightarrow \text{obj}(\mu\nu\text{Poly}_{\mathcal{A}})$ and $\text{codom} : \overline{\times G} \rightarrow \text{obj}(\mu\nu\text{Poly}_{\mathcal{B}})$ give, respectively, the domain and codomain of each functor in $\overline{\times G}$.

Since G creates initial algebras and terminal coalgebras, it is enough to show that, for any $\mu\nu$ -polynomial $H : \mathcal{C} \rightarrow \mathcal{D}$ in $\mu\nu\text{Poly}_{\mathcal{A}}$, there is a morphism \bar{H} of $\mu\nu\text{Poly}_{\mathcal{B}}$ such that there is an isomorphism:

$$\begin{array}{ccc}
 \mathcal{C} & \xrightarrow{H} & \mathcal{D} \\
 \text{dom}^{-1}(\mathcal{C}) \downarrow & \cong & \downarrow \text{dom}^{-1}(\mathcal{D}) \\
 \underline{\mathcal{C}} & \xrightarrow{\underline{H}} & \underline{\mathcal{D}}
 \end{array}
 \tag{113}$$

where $\underline{\mathcal{D}} := \text{codom} \circ \text{dom}^{-1}(\mathcal{D})$ and $\underline{\mathcal{C}} := \text{codom} \circ \text{dom}^{-1}(\mathcal{C})$.

We start by proving that the objects of $\mu\nu\text{Poly}_{\mathcal{A}}$ together with the functors that satisfy the property above do form a subcategory of **Cat**. Indeed, observe that the identities do satisfy the condition above, since it is always true that

$$\text{id}_{\underline{\mathcal{C}}} \circ \text{dom}^{-1}(\mathcal{C}) = \text{dom}^{-1}(\mathcal{C}) \circ \text{id}_{\mathcal{C}}$$

for any given object \mathcal{C} of $\mu\nu\text{Poly}_{\mathcal{A}}$. Moreover, given morphisms $J : \mathcal{D}'' \rightarrow \mathcal{D}'''$ and $E : \mathcal{D}' \rightarrow \mathcal{D}''$ of $\mu\nu\text{Poly}_{\mathcal{A}}$ such that we have natural isomorphisms:

$$\begin{aligned}
 \gamma : \underline{E} \circ \text{dom}^{-1}(\mathcal{D}') &\cong \text{dom}^{-1}(\mathcal{D}'') \circ \underline{E} \\
 \gamma' : \underline{J} \circ \text{dom}^{-1}(\mathcal{D}'') &\cong \text{dom}^{-1}(\mathcal{D}''') \circ \underline{J}
 \end{aligned}$$

in which \underline{J} and \underline{E} are morphisms of $\mu\nu\text{Poly}_{\mathcal{B}}$, we have that

$$\begin{array}{ccccc}
 \mathcal{D}' & \xrightarrow{E} & \mathcal{D}'' & \xrightarrow{J} & \mathcal{D}''' \\
 \text{dom}^{-1}(\mathcal{D}') \downarrow & \gamma \swarrow & \downarrow \text{dom}^{-1}(\mathcal{D}'') & \swarrow \gamma' & \downarrow \text{dom}^{-1}(\mathcal{D}''') \\
 \underline{\mathcal{D}'} & \xrightarrow{\underline{E}} & \underline{\mathcal{D}''} & \xrightarrow{\underline{J}} & \underline{\mathcal{D}'''}
 \end{array}
 \tag{114}$$

is a natural isomorphism and $\underline{J} \circ \underline{E}$ is a morphism in $\mu\nu\text{Poly}_{\mathcal{B}}$.

Finally, we complete the proof that all the morphisms of $\mu\nu\text{Poly}_{\mathcal{A}}$ satisfy the property above by proving by induction over the Definition 6 of $\mu\nu\text{Poly}_{\mathcal{A}}$.

(M1) for any object \mathcal{C} of $\mu\nu\text{Poly}_{\mathcal{A}}$, the unique functor $\mathcal{C} \rightarrow \mathbb{1}$ is such that

$$\begin{array}{ccc}
 \mathcal{C} & \longrightarrow & \mathbb{1} \\
 \text{dom}^{-1}(\mathcal{C}) \downarrow & & \downarrow \text{dom}^{-1}(\mathbb{1}) \\
 \underline{\mathcal{C}} & \longrightarrow & \mathbb{1}
 \end{array}
 \tag{115}$$

commutes and, of course, $\underline{\mathcal{C}} \rightarrow \mathbb{1}$ is a morphism in $\mu\nu\text{Poly}_{\mathcal{B}}$;

(M2) for any object \mathcal{D} of $\mu\nu\text{Poly}_{\mathcal{A}}$, given a functor $W : \mathbb{1} \rightarrow \mathcal{D}$ (which belongs to $\mu\nu\text{Poly}_{\mathcal{A}}$), we have that $\text{dom}^{-1}(\mathcal{D}) \circ W$ is a morphism of $\mu\nu\text{Poly}_{\mathcal{B}}$ such that

$$\begin{array}{ccc}
 \mathbb{1} & \xrightarrow{W} & \mathcal{D} \\
 \text{dom}^{-1}(\mathbb{1}) \downarrow & & \downarrow \text{dom}^{-1}(\mathcal{D}) \\
 \mathbb{1} & \xrightarrow{\text{dom}^{-1}(\mathcal{D}) \circ W} & \underline{\mathcal{D}}
 \end{array}
 \tag{116}$$

commutes:

(M3) consider the binary product $\times : \mathcal{A} \times \mathcal{A} \rightarrow \mathcal{A}$ (which exists, since G is monadic). We have that $\times : \mathcal{B} \times \mathcal{B} \rightarrow \mathcal{B}$ (which is a morphism of $\mu\nu\text{Poly}_{\mathcal{B}}$) is such that we have an isomorphism:

$$\begin{array}{ccc}
 \mathcal{A} \times \mathcal{A} & \xrightarrow{\quad \times \quad} & \mathcal{A} \\
 \text{dom}^{-1}(\mathcal{A} \times \mathcal{A}) \downarrow & \cong & \downarrow \text{dom}^{-1}(\mathcal{A}) \\
 \mathcal{B} \times \mathcal{B} & \xrightarrow{\quad \times \quad} & \mathcal{B}
 \end{array}
 \tag{117}$$

since $G : \mathcal{A} \rightarrow \mathcal{B}$ preserves products and

$$\text{dom}^{-1}(\mathcal{A}) = G, \quad \text{dom}^{-1}(\mathcal{A} \times \mathcal{A}) = G \times G;$$

(M4) consider the binary coproduct $\sqcup : \mathcal{A} \times \mathcal{A} \rightarrow \mathcal{A}$ (which exists, since G is comonadic). We have that $\sqcup : \mathcal{B} \times \mathcal{B} \rightarrow \mathcal{B}$ (which is a morphism of $\mu\nu\text{Poly}_{\mathcal{B}}$) is such that we have an isomorphism:

$$\begin{array}{ccc}
 \mathcal{A} \times \mathcal{A} & \xrightarrow{\quad \sqcup \quad} & \mathcal{A} \\
 G \times G = \text{dom}^{-1}(\mathcal{A} \times \mathcal{A}) \downarrow & \cong & \downarrow G = \text{dom}^{-1}(\mathcal{A}) \\
 \mathcal{B} \times \mathcal{B} & \xrightarrow{\quad \sqcup \quad} & \mathcal{B}
 \end{array}
 \tag{118}$$

since $G : \mathcal{A} \rightarrow \mathcal{B}$ preserves coproducts.

(M5) for any pair of objects $(\mathcal{C}, \mathcal{D}) \in \mu\nu\text{Poly}_{\mathcal{A}} \times \mu\nu\text{Poly}_{\mathcal{A}}$, we have, of course, that

$$\begin{array}{ccc}
 \mathcal{C} \times \mathcal{D} \xrightarrow{\pi_1} \mathcal{C} & & \mathcal{C} \times \mathcal{D} \xrightarrow{\pi_2} \mathcal{D} \\
 \text{dom}^{-1}(\mathcal{C} \times \mathcal{D}) \downarrow & & \downarrow \text{dom}^{-1}(\mathcal{C}) \quad \text{dom}^{-1}(\mathcal{C} \times \mathcal{D}) \downarrow & & \downarrow \text{dom}^{-1}(\mathcal{D}) \\
 \bar{\mathcal{C}} \times \bar{\mathcal{D}} \xrightarrow{\pi_1} \bar{\mathcal{C}} & & \bar{\mathcal{C}} \times \bar{\mathcal{D}} \xrightarrow{\pi_2} \bar{\mathcal{D}}
 \end{array}
 \tag{119}$$

commute and $\pi_1 : \bar{\mathcal{C}} \times \bar{\mathcal{D}} \rightarrow \bar{\mathcal{C}}$ and $\pi_2 : \bar{\mathcal{C}} \times \bar{\mathcal{D}} \rightarrow \bar{\mathcal{D}}$ are morphisms in $\mu\nu\text{Poly}_{\mathcal{B}}$.

(M6) given objects $\mathcal{D}', \mathcal{D}'', \mathcal{D}'''$ of $\mu\nu\text{Poly}_{\mathcal{A}}$, if $E : \mathcal{D}' \rightarrow \mathcal{D}''$ and $J : \mathcal{D}' \rightarrow \mathcal{D}'''$ are morphisms of $\mu\nu\text{Poly}_{\mathcal{A}}$ such that we have natural isomorphisms:

$$\begin{aligned}
 \gamma : \bar{E} \circ \text{dom}^{-1}(\mathcal{D}') &\cong \text{dom}^{-1}(\mathcal{D}'') \circ E \\
 \gamma' : \bar{J} \circ \text{dom}^{-1}(\mathcal{D}') &\cong \text{dom}^{-1}(\mathcal{D}''') \circ J
 \end{aligned}$$

in which \bar{J} and \bar{E} are morphisms of $\mu\nu\text{Poly}_{\mathcal{B}}$, then (\bar{E}, \bar{J}) is a morphism in $\mu\nu\text{Poly}_{\mathcal{B}}$ and (γ, γ') defines an isomorphism:

$$(\bar{E}, \bar{J}) \circ \text{dom}^{-1}(\mathcal{D}') \cong \text{dom}^{-1}(\mathcal{D}'' \times \mathcal{D}''') \circ (E, J).
 \tag{120}$$

(M7) if \mathcal{C} is an object of $\mu\nu\text{Poly}_{\mathcal{A}}$ and $H : \mathcal{C} \times \mathcal{A} \rightarrow \mathcal{A}$ is a morphism of $\mu\nu\text{Poly}_{\mathcal{A}}$ such that there is an isomorphism:

$$\gamma : \bar{H} \circ \text{dom}^{-1}(\mathcal{C} \times \mathcal{A}) \cong \text{dom}^{-1}(\mathcal{A}) \circ H$$

in which \bar{H} is a morphism of $\mu\nu\text{Poly}_{\mathcal{B}}$, then, since G creates initial algebras and terminal coalgebras, we get that there are natural transformations:

$$\mu \bar{H} \circ \text{dom}^{-1}(\mathcal{A}) \cong \text{dom}^{-1}(\mathcal{A}) \circ \mu H$$

$$v\overline{H} \circ \text{dom}^{-1}(\mathcal{A}) \cong \text{dom}^{-1}(\mathcal{A}) \circ vH$$

and, of course, $\mu\overline{H}$ and $v\overline{H}$ are morphisms of $\mu\nu\text{Poly}_{\mathcal{B}}$. □

11.5 $\mu\nu$ -polynomials in product categories

Before applying the results above to study the $\mu\nu$ -polynomials in suitable scones $\mathcal{D} \downarrow G$, we need to study the $\mu\nu$ -polynomials in product categories $\mathcal{C} \times \mathcal{D}$. We start by showing that:

Lemma 111. *Let $(E_i : \mathcal{C}_i \rightarrow \mathcal{C}_i)_{i \in L}$ be a (possibly infinite) family of endofunctors such that E_i has initial algebra $(\mu E_i, \text{in}_{E_i})$ and terminal coalgebra $(\nu E_i, \text{out}_{E_i})$. The functor defined by the product:*

$$\prod_{i \in L} E_i : \prod_{i \in L} \mathcal{C}_i \rightarrow \prod_{i \in L} \mathcal{C}_i \tag{121}$$

has initial algebra given by $(\mu E_i, \text{in}_{E_i})_{i \in L}$ and terminal coalgebra given by $(\nu E_i, \text{out}_{E_i})_{i \in L}$.

As a consequence, if $(H_i : \mathcal{A}_i \times \mathcal{C}_i \rightarrow \mathcal{C}_i)_{i \in L}$ is a (possibly infinite) family of functors with parameterized initial algebras and terminal coalgebras, then $\prod_{i \in L} H_i$ has parameterized initial algebra given by $\prod_{i \in L} \mu H_i : \prod_{i \in L} \mathcal{A}_i \rightarrow \prod_{i \in L} \mathcal{C}_i$ and parameterized terminal coalgebra given by $\prod_{i \in L} \nu H_i : \prod_{i \in L} \mathcal{A}_i \rightarrow \prod_{i \in L} \mathcal{C}_i$.

Proof. Given an $\left(\prod_{i \in L} E_i\right)$ -algebra $(Y_i, \xi_i)_{i \in L}$, we have that (Y_i, ξ_i) is an E_i -algebra for every $i \in L$. Therefore, by the universal property of $(\mu E_i, \text{in}_{E_i})$ for each i , we conclude that

$$\text{fold}_{\prod_{i \in L} E_i} (Y_i, \xi_i)_{i \in L} := (\text{fold}_{E_i} (Y_i, \xi_i))_{i \in L} \tag{122}$$

is the unique morphism in $\prod_{i \in L} \mathcal{C}_i$ such that

$$\begin{array}{ccc}
 \prod_{i \in L} E_i (\mu E_i)_{i \in L} = (E_i (\mu E_i))_{i \in L} & \xrightarrow{\prod_{i \in L} E_i (\text{fold}_{E_i} (Y_i, \xi_i))_{i \in L}} & \prod_{i \in L} E_i (Y_i)_{i \in L} \\
 \downarrow (\text{in}_{E_i})_{i \in L} & \prod_{i \in L} E_i \left(\text{fold}_{\prod_{i \in L} E_i} (Y_i, \xi_i)_{i \in L} \right) & \downarrow (\xi_i)_{i \in L} \\
 (\mu E_i)_{i \in L} & \xrightarrow{\text{fold}_{\prod_{i \in L} E_i} (Y_i, \xi_i)_{i \in L}} & (Y_i)_{i \in L} \\
 & (\text{fold}_{E_i} (Y_i, \xi_i))_{i \in L} &
 \end{array} \tag{123}$$

holds. This proves that $(\mu E_i, \text{in}_{E_i})_{i \in L}$ is the initial $\left(\prod_{i \in L} E_i\right)$ -algebra. Dually, $(\nu E_i, \text{out}_{E_i})_{i \in L}$ is the terminal $\left(\prod_{i \in L} E_i\right)$ -coalgebra. □

We prove below that the binary product of categories with $\mu\nu$ -polynomials has $\mu\nu$ -polynomials. We start by:

Definition 112 ($\text{deck}_{\mathcal{A}}$). Let $(C_i)_{i \in L}$ be a (possibly infinite) family of categories. We establish a family:

$$(\text{deck}_{\mathcal{A}}^i : \mathcal{A} \rightarrow \text{deck}_i(\mathcal{A}))_{(\mathcal{A}, i) \in \left(\text{obj} \left(\mu\nu\text{Poly}_{\prod_{i \in L} C_i} \right) \times L \right)} \tag{124}$$

of functors, where $\text{deck}_i(\mathcal{A}) \in \text{obj}(\mu\nu\text{Poly}_{C_i})$, by induction on the objects of $\mu\nu\text{Poly}_{\prod_{i \in L} C_i}$:

$$\text{deck}_{\mathbb{1}}^i := \text{id}_{\mathbb{1}}; \tag{125} \quad \text{deck}_{\prod_{i \in L} C_i}^i := \pi_{C_i} : \prod_{i \in L} C_i \rightarrow C_i; \tag{126}$$

$$\text{deck}_{\mathcal{A} \times \mathcal{A}'}^i := \text{deck}_{\mathcal{A}}^i \times \text{deck}_{\mathcal{A}'}^i, \text{ if } (\mathcal{A}, \mathcal{A}') \in \text{obj} \left(\mu\nu\text{Poly}_{\prod_{i \in L} C_i} \right)^2. \tag{127}$$

Finally, for each $\mathcal{A} \in \text{obj} \left(\mu\nu\text{Poly}_{\prod_{i \in L} C_i} \right)$, we define the isomorphism of categories:

$$\text{deck}_{\mathcal{A}} := (\text{deck}_{\mathcal{A}}^0, \text{deck}_{\mathcal{A}}^1). \tag{128}$$

Lemma 113. Let $(C_i)_{i \in L}$ be a (possibly infinite) family of categories with $\mu\nu$ -polynomials. For each pair $(\mathcal{A}, \mathcal{A}') \in \text{obj} \left(\mu\nu\text{Poly}_{\prod_{i \in L} C_i} \right)^2$ and any functor $H : \mathcal{A} \rightarrow \mathcal{A}'$ in $\mu\nu\text{Poly}_{\prod_{i \in L} C_i}$, we have that $\text{deck}_{\mathcal{A}'} \circ H \circ \text{deck}_{\mathcal{A}}^{-1} = \prod_{i \in L} H_i$ for some morphism $(H_i)_{i \in L}$ in $\prod_{i \in L} \mu\nu\text{Poly}_{C_i}$.

Proof. It is clear the property above is closed under composition, and the identity on $\prod_{i \in L} C_i$ satisfies the property. Moreover, for the base case (see Definition 6), it is clear that the functors in $\mu\nu\text{Poly}_{\prod_{i \in L} C_i}$ defined by the base cases (M1) and (M2) satisfy the statement above. Moreover, since the binary products and coproducts in $\prod_{i \in L} C_i$ are defined pointwise, it is also true that (M3) and (M4) satisfy the statement above. Finally, it is also clear that the statement above holds for (M5) and (M6).

We assume, by induction, that $H : \mathcal{A} \times \prod_{i \in L} C_i \rightarrow \prod_{i \in L} C_i$ is a morphism of $\mu\nu\text{Poly}_{\prod_{i \in L} C_i}$ such that $H \circ \text{deck}_{\mathcal{A} \times \prod_{i \in L} C_i}^{-1} = \prod_{i \in L} H_i$ for some morphism $(H_i)_{i \in L}$ in $\prod_{i \in L} \mu\nu\text{Poly}_{C_i}$.

Since C_i has $\mu\nu$ -polynomials for all $i \in L$, we have that H_i has parameterized initial algebras and parameterized terminal coalgebras for all $i \in L$. Therefore, $\prod_{i \in L} H_i$ has parameterized initial algebra

$\prod_{i \in L} \mu H_i$ and parameterized terminal coalgebra $\prod_{i \in L} \nu H_i$ by Lemma 111. Hence, $\mu H = \left(\prod_{i \in L} \mu H_i \right) \circ$

$\text{deck}_{\mathcal{A}}$ and $\nu H = \left(\prod_{i \in L} \nu H_i \right) \circ \text{deck}_{\mathcal{A}}$ where $(\mu H_i)_{i \in L}$ and $(\nu H_i)_{i \in L}$ are morphisms in $\prod_{i \in L} \mu\nu\text{Poly}_{C_i}$.

This completes the proof. □

Theorem 114. Let $(C_i)_{i \in L}$ be a (possibly infinite) family of categories with $\mu\nu$ -polynomials. The category $\prod_{i \in L} C_i$ has $\mu\nu$ -polynomials.

Proof. For each endofunctor $E: \prod_{i \in L} \mathcal{C}_i \rightarrow \prod_{i \in L} \mathcal{C}_i$ in $\mu\nu\text{Poly}_{\mathcal{C} \times \mathcal{D}}$, we conclude that $E = \prod_{i \in L} E_i$ for some morphism $(E_i: \mathcal{C}_i \rightarrow \mathcal{C}_i)_{i \in L}$ of $\mu\nu\text{Poly}_{\prod_{i \in L} \mathcal{C}_i}$ by Lemma 113. Therefore, by Lemma 111, E has initial algebra and terminal coalgebra, since the functors of the family $(E_i: \mathcal{C}_i \rightarrow \mathcal{C}_i)_{i \in L}$ do. \square

11.6 Suitable scones have $\mu\nu$ -polynomials

Finally, we establish the existence of $\mu\nu$ -polynomials in the scone, and the preservation of the initial algebras and terminal coalgebras by the forgetful functor.

Corollary 115. *Let \mathcal{C} and \mathcal{D} be categories with $\mu\nu$ -polynomials. If $G: \mathcal{C} \rightarrow \mathcal{D}$ has a left adjoint, then $\mathcal{D} \downarrow G$ has $\mu\nu$ -polynomials and*

$$L: \mathcal{D} \downarrow G \rightarrow \mathcal{D} \times \mathcal{C} \tag{129}$$

(strictly) preserves (in fact, creates) $\mu\nu$ -polynomials.

Proof. By Corollary 99, we have that L is monadic and comonadic. Hence, it creates $\mu\nu$ -polynomials and we get the conclusion of the result provided that $\mathcal{D} \times \mathcal{C}$ has $\mu\nu$ -polynomials.

Indeed, by Theorem 114, $\mathcal{D} \times \mathcal{C}$ has $\mu\nu$ -polynomials provided that \mathcal{D} and \mathcal{C} have $\mu\nu$ -polynomials. \square

11.7 The projection $\mathcal{D} \downarrow G \rightarrow \mathcal{C}$

Let \mathcal{C} and \mathcal{D} be bicartesian closed categories with finite limits. Recall that $\pi_{\mathcal{C}}: \mathcal{D} \times \mathcal{C} \rightarrow \mathcal{C}$ has left and right adjoints, respectively, given by $W \mapsto (W, 0)$ and $W \mapsto (W, \mathbb{1})$. Therefore, assuming that $G: \mathcal{C} \rightarrow \mathcal{D}$ has a left adjoint, we get that

$$\mathcal{D} \downarrow G \xrightarrow{L} \mathcal{D} \times \mathcal{C} \xrightarrow{\pi_2} \mathcal{C} \tag{130}$$

has a left adjoint and a right adjoint. Therefore, it preserves limits, colimits, initial algebras, and terminal coalgebras. Finally, (130) preserves the closed structure by Corollary (104).

Corollary 116. *Let \mathcal{C} and \mathcal{D} be finitely complete bicartesian closed categories that have $\mu\nu$ -polynomials. If $G: \mathcal{C} \rightarrow \mathcal{D}$ has a left adjoint, the category $\mathcal{D} \downarrow G$ is a finitely complete bicartesian closed category with $\mu\nu$ -polynomials, and (130) is a (strictly) bicartesian closed functor that (strictly) preserves $\mu\nu$ -polynomials.*

Furthermore, if, additionally, \mathcal{C} and \mathcal{D} have infinite coproducts, so does $\mathcal{D} \downarrow G$ and (130) (strictly) preserves them.

12. Correctness of CHAD for Tuples and Variant Types, by Logical Relations

Henceforth, we assume the hypothesis established in Section 10.6 and rely on the concrete semantics and notation established in Section 10.

In this section, we present the basic correctness theorem for tuples and variant types, which serves as a crucial step toward establishing the full correctness theorem for data types. More precisely, we prove:

Theorem 117 (Correctness of CHAD for tuples and variant tuples). *For any well-typed program,*

$$x: \tau \vdash t: \sigma,$$

where τ, σ are data types that do not involve inductive types, we have that $\llbracket t \rrbracket$ is differentiable. Moreover, (131) and (132) hold

$$\Sigma \llbracket \overrightarrow{\mathcal{D}}(t) \rrbracket = \mathcal{D} \llbracket t \rrbracket \tag{131} \qquad \qquad \qquad \overset{t}{\Sigma} \llbracket \overleftarrow{\mathcal{D}}(t) \rrbracket = \mathcal{D}^t \llbracket t \rrbracket \tag{132}$$

It should be noted that: (1) we prove our result only assuming that the semantics of the primitive operations are differentiable instead of requiring them to be smooth;¹¹ (2) t above might, in particular, have subprograms that use higher-order functions and (co)inductive types.

The argument we present below is a categorical version of a semantic open logical relations proof; see, for instance, Barthe et al. (2020), Huot et al. (2020), Vákár (2021), and Vákár and Smeding (2022). We follow the perspective described in Lucatelli Nunes and Vákár (2022b, Section 4) and Lucatelli Nunes and Vákár (2022a).

The precise statement Theorem 117 is presented in Theorem 124.

12.1 The score for the correctness proof

We first establish the appropriate score for our proof (see Section 11).

By Proposition 81, Corollaries 82 and 83, we conclude, in particular, that $\mathbf{Fam}(\mathbf{Set})$, $\mathbf{Fam}(\mathbf{Vect})$ and $\mathbf{Fam}(\mathbf{Vect}^{\text{op}})$ are finitely complete cartesian closed categories with $\mu\nu$ -polynomials and infinite coproducts. Therefore, we conclude that $\mathbf{Fam}(\mathbf{Set}) \times \mathbf{Fam}(\mathbf{Vect}) \times \mathbf{Fam}(\mathbf{Vect}^{\text{op}})$ is a finitely complete cartesian closed category with $\mu\nu$ -polynomials and infinite coproducts: see Theorem 114 for the result on $\mu\nu$ -polynomials.

We consider the score along (133), which is representable by the coproduct

$$\coprod_{k \in \mathbb{N}} \left(\mathbb{R}^k, \left(\mathbb{R}^k, \underline{\mathbb{R}}^k \right), \left(\mathbb{R}^k, \underline{\mathbb{R}}^k \right) \right) \text{ in } \mathbf{Fam}(\mathbf{Set}) \times \mathbf{Fam}(\mathbf{Vect}) \times \mathbf{Fam}(\mathbf{Vect}^{\text{op}}):$$

$$\begin{aligned} \overleftrightarrow{G} &: \mathbf{Fam}(\mathbf{Set}) \times \mathbf{Fam}(\mathbf{Vect}) \rightarrow \mathbf{Set} \tag{133} \\ \overleftrightarrow{G} &:= \prod_{k \in \mathbb{N}} \left(\mathbf{Fam}(\mathbf{Set}) \times \mathbf{Fam}(\mathbf{Vect}) \times \mathbf{Fam}(\mathbf{Vect}^{\text{op}}) \left(\left(\mathbb{R}^k, \left(\mathbb{R}^k, \underline{\mathbb{R}}^k \right), \left(\mathbb{R}^k, \underline{\mathbb{R}}^k \right) \right), - \right) \right) \end{aligned}$$

Moreover, (134) given by the copower in $\mathbf{Fam}(\mathbf{Set}) \times \mathbf{Fam}(\mathbf{Vect}) \times \mathbf{Fam}(\mathbf{Vect}^{\text{op}})$ defines the left adjoint $\overleftarrow{F} \dashv \overleftrightarrow{G}$. As a consequence, we get Theorem 118 by Corollary 116:

$$\begin{aligned} \overleftarrow{F} &: \mathbf{Set} \rightarrow \mathbf{Fam}(\mathbf{Set}) \times \mathbf{Fam}(\mathbf{Vect}) \times \mathbf{Fam}(\mathbf{Vect}^{\text{op}}) \tag{134} \\ W &\mapsto W \otimes \prod_{k \in \mathbb{N}} \left(\mathbb{R}^k, \left(\mathbb{R}^k, \underline{\mathbb{R}}^k \right), \left(\mathbb{R}^k, \underline{\mathbb{R}}^k \right) \right) \cong \prod_{x \in W} \left(\prod_{k \in \mathbb{N}} \left(\mathbb{R}^k, \left(\mathbb{R}^k, \underline{\mathbb{R}}^k \right), \left(\mathbb{R}^k, \underline{\mathbb{R}}^k \right) \right) \right) \end{aligned}$$

Theorem 118. $\mathbf{Set} \downarrow \overleftrightarrow{G}$ is a finitely complete cartesian closed categories with $\mu\nu$ -polynomials and infinite coproducts. Moreover, (135) is a strictly bicartesian closed functor that preserves $\mu\nu$ -polynomials and (infinite) coproducts:

$$\begin{aligned} \mathbf{Set} \downarrow \overleftrightarrow{G} &\rightarrow \mathbf{Set} \times \mathbf{Fam}(\mathbf{Set}) \times \mathbf{Fam}(\mathbf{Vect}) \times \mathbf{Fam}(\mathbf{Vect}^{\text{op}}) \rightarrow \mathbf{Fam}(\mathbf{Set}) \\ &\times \mathbf{Fam}(\mathbf{Vect}) \times \mathbf{Fam}(\mathbf{Vect}^{\text{op}}) \end{aligned} \tag{135}$$

Definition 119 (Score). For short, we henceforth denote by (136), where $\overleftarrow{\mathbf{Score}} := \mathbf{Set} \downarrow \overleftrightarrow{G}$, the forgetful functor (135):

$$\overleftarrow{\pi} : \overleftarrow{\mathbf{Score}} \rightarrow \mathbf{Fam}(\mathbf{Set}) \times \mathbf{Fam}(\mathbf{Vect}) \times \mathbf{Fam}(\mathbf{Vect}^{\text{op}}) \tag{136}$$

12.2 The logical relations

Guided by the characterization of differentiable morphisms and their derivatives (Lemma 94), we now define the objects in $\overleftarrow{\mathbf{Scone}}$ that will provide us with the appropriate predicates for our logical relations argument.

It should be noted that, for any object $(Y, (W, w), (Z, z))$ in $\mathbf{Fam}(\mathbf{Set}) \times \mathbf{Fam}(\mathbf{Vect}) \times \mathbf{Fam}(\mathbf{Vect}^{\text{op}})$, the elements of $\overleftarrow{G}(Y, (W, w), (Z, z))$ are families $(f_k, g_k, h_k)_{k \in \mathbb{N}}$ where, for each $k \in \mathbb{N}$, $f_k : \mathbb{R}^k \rightarrow Y$ is a morphism in $\mathbf{Fam}(\mathbf{Set})$, $g_k : (\mathbb{R}^k, \underline{\mathbb{R}}^k) \rightarrow (W, w)$ is a morphism in $\mathbf{Fam}(\mathbf{Vect})$ and $h_k : (\mathbb{R}^k, \underline{\mathbb{R}}^k) \rightarrow (Z, z)$ is a morphism in $\mathbf{Fam}(\mathbf{Vect}^{\text{op}})$.

Definition 120 ($\overleftarrow{\llbracket \mathbf{real}^n \rrbracket}$). For each n -dimensional array $\mathbf{real}^n \in \mathbf{Syn}$, we define the subset (137) of $\overleftarrow{\mathbb{R}}^n := \overleftarrow{G}(\mathbb{R}^n, (\mathbb{R}^n, \underline{\mathbb{R}}^n), (\mathbb{R}^n, \underline{\mathbb{R}}^n))$:

$$\overleftarrow{\llbracket \mathbf{real}^n \rrbracket} := \left\{ (f_k, g_k, h_k)_{k \in \mathbb{N}} \in \overleftarrow{\mathbb{R}}^n : \forall k \in \mathbb{N}, f_k \text{ is differentiable, } g_k = \mathcal{D}f_k, h_k = \mathcal{D}^t f_k \right\} \tag{137}$$

Denoting the subset inclusion by:

$$\text{inc} : \overleftarrow{\llbracket \mathbf{real}^n \rrbracket} \rightarrow \overleftarrow{G}(\mathbb{R}^n, (\mathbb{R}^n, \underline{\mathbb{R}}^n), (\mathbb{R}^n, \underline{\mathbb{R}}^n)),$$

we define the object (138) of $\overleftarrow{\mathbf{Scone}}$:

$$\overleftarrow{\llbracket \mathbf{real}^n \rrbracket} := \left(\overleftarrow{\llbracket \mathbf{real}^n \rrbracket}, (\mathbb{R}^n, (\mathbb{R}^n, \underline{\mathbb{R}}^n), (\mathbb{R}^n, \underline{\mathbb{R}}^n)), \text{inc} \right). \tag{138}$$

Recall that we denote by \mathfrak{E} the set of Euclidean families defined in (87). Theorem 121 relies on the canonical diffeomorphisms given in Definition 92.

Theorem 121. Let (f, g, h) be a morphism in $\mathbf{Fam}(\mathbf{Set}) \times \mathbf{Fam}(\mathbf{Vect}) \times \mathbf{Fam}(\mathbf{Vect}^{\text{op}})$. Assuming that $f : A \rightarrow B$ is such that A and B are Euclidean families, we have that (i) implies (ii).

(i) There is a morphism:

$$\alpha : \coprod_{j \in J} \left(\prod_{i=1}^{n_j} \overleftarrow{\llbracket \mathbf{real}^{q_{(j,i)}} \rrbracket} \right) \rightarrow \coprod_{l \in L} \left(\prod_{t=1}^{m_l} \overleftarrow{\llbracket \mathbf{real}^{s_{(l,t)}} \rrbracket} \right) \tag{139}$$

in $\overleftarrow{\mathbf{Scone}}$, where $(n_j)_{j \in J}, (m_l)_{l \in L}, ((q_{(j,i)})_{i \in \{1, \dots, n_j\}})_{j \in J}$ and $((s_{(l,t)})_{t \in \{1, \dots, m_l\}})_{l \in L}$ are (possibly infinite) families of natural numbers, such that

$$\overleftarrow{\pi}(\alpha) = \left(\epsilon_{n_B} \circ f \circ \epsilon_{n_A}^{-1}, \mathcal{D}(\epsilon_{n_B}) \circ g \circ \mathcal{D}(\epsilon_{n_A})^{-1}, \mathcal{D}^t(\epsilon_{n_B}) \circ h \circ \mathcal{D}^t(\epsilon_{n_A})^{-1} \right). \tag{140}$$

(ii) The morphism f is differentiable, $\mathcal{D}f = g$ and $\mathcal{D}^t f = h$.

Proof. We start by establishing the objects \mathfrak{S}_0 and \mathfrak{S}_1 of $\overleftarrow{\mathbf{Scone}}$ together with the canonical isomorphisms (147) and (148).

Let $q_j := \sum_{i=1}^{n_j} q_{(j,i)}$ and $s_l := \sum_{t=1}^{m_l} s_{(l,t)}$. We define the objects \mathfrak{A}_0 and \mathfrak{A}_1 of $\mathbf{Set} \times \mathbf{Fam}(\mathbf{Vect}) \times$

$\mathbf{Fam}(\mathbf{Vect}^{\text{op}})$ by (141) and (142): the construction of infinite coproducts in $\overleftarrow{\mathbf{Scone}}$ follows from Section 11.1:

$$\begin{aligned} \mathfrak{A}_0 &:= \coprod_{j \in J} (\mathbb{R}^{q_j}, (\mathbb{R}^{q_j}, \underline{\mathbb{R}}^{q_j}), (\mathbb{R}^{q_j}, \underline{\mathbb{R}}^{q_j})) \\ &= \left(\coprod_{j \in J} \mathbb{R}^{q_j}, \left(\coprod_{j \in J} \mathbb{R}^{q_j}, \langle \underline{\mathbb{R}}^{q_j} \rangle_{j \in J} \right), \left(\coprod_{j \in J} \mathbb{R}^{q_j}, \langle \underline{\mathbb{R}}^{q_j} \rangle_{j \in J} \right) \right) \end{aligned} \tag{141}$$

$$\begin{aligned} \mathfrak{A}_1 &:= \coprod_{l \in L} (\mathbb{R}^{s_l}, (\mathbb{R}^{s_l}, \underline{\mathbb{R}}^{s_l}), (\mathbb{R}^{s_l}, \underline{\mathbb{R}}^{s_l})) \\ &= \left(\coprod_{l \in L} \mathbb{R}^{s_l}, \left(\coprod_{l \in L} \mathbb{R}^{s_l}, \langle \underline{\mathbb{R}}^{s_l} \rangle_{l \in L} \right), \left(\coprod_{l \in L} \mathbb{R}^{s_l}, \langle \underline{\mathbb{R}}^{s_l} \rangle_{l \in L} \right) \right) \end{aligned} \tag{142}$$

We consider the subsets $\overleftarrow{\mathfrak{G}}_0 \subset \overleftarrow{\mathfrak{G}}(\mathfrak{A}_0)$ and $\overleftarrow{\mathfrak{G}}_1 \subset \overleftarrow{\mathfrak{G}}(\mathfrak{A}_1)$ defined by (143) and (144). Denoting by inc the appropriate subset inclusions, we define the objects $\mathfrak{S}_0 := (\overleftarrow{\mathfrak{G}}_0, \mathfrak{A}_0, \text{inc})$ and $\mathfrak{S}_1 := (\overleftarrow{\mathfrak{G}}_1, \mathfrak{A}_1, \text{inc})$ of $\overleftarrow{\mathbf{Scone}}$:

$$\overleftarrow{\mathfrak{G}}_0 := \left\{ (f_k, g_k, h_k)_{k \in \mathbb{N}} \in \overleftarrow{\mathfrak{G}}(\mathfrak{A}_0) : \forall k \in \mathbb{N}, f_k \text{ is differentiable, } g_k = \mathfrak{D}f_k, h_k = \mathfrak{D}^t f_k \right\} \tag{143}$$

$$\overleftarrow{\mathfrak{G}}_1 := \left\{ (f_k, g_k, h_k)_{k \in \mathbb{N}} \in \overleftarrow{\mathfrak{G}}(\mathfrak{A}_1) : \forall k \in \mathbb{N}, f_k \text{ is differentiable, } g_k = \mathfrak{D}f_k, h_k = \mathfrak{D}^t f_k \right\} \tag{144}$$

By the results of Section 11.1, the chain rule (Lemma 90) and Definition 120, since the canonical isomorphisms (145) and (146) are diffeomorphisms, there are (invertible) functions $\underline{\text{can}}_0$ and $\underline{\text{can}}_1$, respectively, induced by the compositions with $(\text{can}_0, \mathfrak{D}(\text{can}_0), \mathfrak{D}^t(\text{can}_0))$ and $(\text{can}_1, \mathfrak{D}(\text{can}_1), \mathfrak{D}^t(\text{can}_1))$, such that (147) and (148) define isomorphisms in $\overleftarrow{\mathbf{Scone}}$:

$$\text{can}_0 : \coprod_{j \in J} \left(\prod_{i=1}^{n_j} \mathbb{R}^{q_{(j,i)}} \right) \xrightarrow{\cong} \coprod_{j \in J} \mathbb{R}^{q_j} \tag{145}$$

$$\text{can}_1 : \coprod_{l \in L} \left(\prod_{t=1}^{m_l} \mathbb{R}^{s_{(l,t)}} \right) \xrightarrow{\cong} \coprod_{l \in L} \mathbb{R}^{s_l} \tag{146}$$

$$\text{c}\tilde{\text{a}}\tilde{\text{n}}_0 := (\underline{\text{can}}_0, (\text{can}_0, \mathfrak{D}(\text{can}_0), \mathfrak{D}^t(\text{can}_0))) : \coprod_{j \in J} \left(\prod_{i=1}^{n_j} \overleftarrow{\mathbb{R}}^{q_{(j,i)}} \right) \xrightarrow{\cong} \mathfrak{S}_0 \tag{147}$$

$$\text{c}\tilde{\text{a}}\tilde{\text{n}}_1 := (\underline{\text{can}}_1, (\text{can}_1, \mathfrak{D}(\text{can}_1), \mathfrak{D}^t(\text{can}_1))) : \coprod_{l \in L} \left(\prod_{t=1}^{m_l} \overleftarrow{\mathbb{R}}^{s_{(l,t)}} \right) \xrightarrow{\cong} \mathfrak{S}_1 \tag{148}$$

Proof of (i) \Rightarrow (ii).

By (i) and chain rule, denoting $f = \epsilon_{n_B} \circ f \circ \epsilon_{n_A}^{-1}$, $g = \mathfrak{D}(\epsilon_{n_B}) \circ g \circ \mathfrak{D}(\epsilon_{n_A})^{-1}$ and $h = \mathfrak{D}^t(\epsilon_{n_B}) \circ h \circ \mathfrak{D}^t(\epsilon_{n_A})^{-1}$, we conclude that there is a morphism α in $\overleftarrow{\mathbf{Scone}}$ such that

$$\begin{aligned} &\overleftarrow{\mathcal{H}} \left(\text{c}\tilde{\text{a}}\tilde{\text{n}}_1 \circ \alpha \circ \text{c}\tilde{\text{a}}\tilde{\text{n}}_0^{-1} \right) \\ &= \\ &\left(\text{can}_1 \circ f \circ \text{can}_0^{-1}, \mathfrak{D}(\text{can}_1) \circ g \circ \mathfrak{D}(\text{can}_0)^{-1}, \mathfrak{D}^t(\text{can}_1) \circ h \circ \mathfrak{D}^t(\text{can}_0)^{-1} \right). \end{aligned}$$

This implies, by the definitions of \mathfrak{S}_0 and \mathfrak{S}_1 , that, for any family $\left(\gamma_k : \mathbb{R}^k \rightarrow \prod_{j \in J} \mathbb{R}^{q_j}\right)_{k \in \mathbb{N}}$ of differentiable functions, we have that, for all $k \in \mathbb{N}$:

- (I) $\text{can}_1 \circ f \circ \text{can}_0^{-1} \circ \gamma_k$ is differentiable,
- (II) $\mathcal{D}(\text{can}_1) \circ g \circ \mathcal{D}(\text{can}_0)^{-1} = \mathcal{D}(\text{can}_1 \circ f \circ \text{can}_0^{-1} \circ \gamma_k)$, and
- (III) $\mathcal{D}^t(\text{can}_1) \circ h \circ \mathcal{D}^t(\text{can}_0)^{-1} = \mathcal{D}^t(\text{can}_1 \circ f \circ \text{can}_0^{-1} \circ \gamma_k)$.

By Lemma 94, this implies that:

- (A) $\text{can}_1 \circ f \circ \text{can}_0^{-1}$ is differentiable,
- (B) $\mathcal{D}(\text{can}_1) \circ g \circ \mathcal{D}(\text{can}_0)^{-1} = \mathcal{D}(\text{can}_1 \circ f \circ \text{can}_0^{-1})$, and
- (C) $\mathcal{D}^t(\text{can}_1) \circ h \circ \mathcal{D}^t(\text{can}_0)^{-1} = \mathcal{D}^t(\text{can}_1 \circ f \circ \text{can}_0^{-1})$.

By the chain rule (Lemma 90) and the fact that can_1 , can_0 , ${}^e n_A$, and ${}^e n_B$ are diffeomorphisms, this implies that f is differentiable, $g = \mathcal{D}f$ and $h = \mathcal{D}^t f$. This completes the proof. \square

12.3 Logical relations as a functor

For each primitive operation $\text{op} \in \text{Op}_{n_1, \dots, n_k}^m$ of the source language, recall that

$$\llbracket \text{op} \rrbracket : \mathbb{R}^{n_1} \times \dots \times \mathbb{R}^{n_k} \rightarrow \mathbb{R}^m$$

is differentiable, $\Sigma \llbracket \vec{\mathcal{D}}(\text{op}) \rrbracket = \mathcal{D} \llbracket \text{op} \rrbracket$, and ${}^t \Sigma \llbracket \overleftarrow{\mathcal{D}}(\text{op}) \rrbracket = \mathcal{D}^t \llbracket \text{op} \rrbracket$ (see Section 10.6). Therefore, for each primitive operation $\text{op} \in \text{Op}_{n_1, \dots, n_l}^m$, we conclude, by the chain rule (Lemma 90), that we can define the morphism:

$$\overleftrightarrow{\llbracket \text{op} \rrbracket} := \left(\llbracket \text{op} \rrbracket, \left(\llbracket \text{op} \rrbracket, \Sigma \llbracket \vec{\mathcal{D}}(\text{op}) \rrbracket, {}^t \Sigma \llbracket \overleftarrow{\mathcal{D}}(\text{op}) \rrbracket \right) \right) : \prod_{i=1}^l \overleftrightarrow{\llbracket \text{real}^{n_i} \rrbracket} \rightarrow \overleftrightarrow{\llbracket \text{real}^m \rrbracket} \tag{149}$$

in $\overleftrightarrow{\mathbf{Scone}}$.

Since \mathbf{Scone} is bicartesian closed and has $\mu\nu$ -polynomials, by the universal property of the category \mathbf{Syn} established in Corollary 15, we conclude:

Lemma 122. *There is a unique strictly bicartesian closed functor:*

$$\overleftrightarrow{\llbracket - \rrbracket} : \mathbf{Syn} \rightarrow \overleftrightarrow{\mathbf{Scone}} \tag{150}$$

that strictly preserves $\mu\nu$ -polynomials such that $\overleftrightarrow{\llbracket - \rrbracket}$ extends the consistent assignment given by (151):

$$\text{real}^n \mapsto \overleftrightarrow{\llbracket \text{real}^n \rrbracket}, \quad \text{op} \mapsto \overleftrightarrow{\llbracket \text{op} \rrbracket}. \tag{151}$$

Let us recall that we defined the forward-mode and reverse-mode CHAD corresponding functors in Corollary 69, which we denote by $\vec{\mathcal{D}}(-)$ and $\overleftarrow{\mathcal{D}}(-)$, respectively. By the universal property of \mathbf{Syn} and the hypothesis established in Section 10.6, we can further conclude that:

Theorem 123 (Correctness commutative diagram). *Diagram (152) commutes*

$$\begin{array}{ccc}
 \mathbf{Syn} & \xrightarrow{(\text{id}, \overrightarrow{\mathcal{D}}(-), \overleftarrow{\mathcal{D}}(-))} & \mathbf{Syn} \times \Sigma_{\mathbf{CSyn}} \mathbf{LSyn} \times \Sigma_{\mathbf{CSyn}} \mathbf{LSyn}^{\text{op}} \\
 \downarrow \llbracket - \rrbracket & & \downarrow \llbracket - \rrbracket \times \Sigma \llbracket - \rrbracket \times \overset{t}{\Sigma} \llbracket - \rrbracket \\
 \mathbf{Scone} & \xrightarrow{\overleftarrow{\pi}} & \mathbf{Fam}(\mathbf{Set}) \times \mathbf{Fam}(\mathbf{Vect}) \times \mathbf{Fam}(\mathbf{Vect}^{\text{op}})
 \end{array} \tag{152}$$

Proof. For each primitive type \mathbf{real}^n and each primitive operation op , we have that Eqs. (153) and (154) hold by CHAD’s soundness for primitives (by which we mean the assumptions of Section 10.6):

$$\overleftarrow{\pi} \left(\overleftarrow{\llbracket \mathbf{real}^n \rrbracket} \right) = (\mathbb{R}^n, (\mathbb{R}^n, \mathbb{R}^n), (\mathbb{R}^n, \mathbb{R}^n)) = \left(\llbracket \mathbf{real}^n \rrbracket, \Sigma \llbracket \overrightarrow{\mathcal{D}}(\mathbf{real}^n) \rrbracket, \overset{t}{\Sigma} \llbracket \mathcal{D}^t(\mathbf{real}^n) \rrbracket \right) \tag{153}$$

$$\overleftarrow{\pi} \left(\overleftarrow{\llbracket \text{op} \rrbracket} \right) = (\llbracket \text{op} \rrbracket, \mathcal{D}(\llbracket \text{op} \rrbracket), \mathcal{D}^t(\llbracket \text{op} \rrbracket)) = \left(\llbracket \text{op} \rrbracket, \Sigma \llbracket \overrightarrow{\mathcal{D}}(\text{op}) \rrbracket, \overset{t}{\Sigma} \llbracket \overleftarrow{\mathcal{D}}(\text{op}) \rrbracket \right) \tag{154}$$

Since $(\llbracket - \rrbracket \times \Sigma \llbracket - \rrbracket \times \overset{t}{\Sigma} \llbracket - \rrbracket) \circ (\text{id}, \overrightarrow{\mathcal{D}}(-), \overleftarrow{\mathcal{D}}(-))$ and $\overleftarrow{\pi} \circ \llbracket - \rrbracket$ are (compositions) of strictly $\mu\nu$ -polynomial-preserving bicartesian closed functors satisfying (153) and (154) for any ground type \mathbf{real}^n and any primitive operation op , we conclude that $(\llbracket - \rrbracket \times \Sigma \llbracket - \rrbracket \times \overset{t}{\Sigma} \llbracket - \rrbracket) \circ (\text{id}, \overrightarrow{\mathcal{D}}(-), \overleftarrow{\mathcal{D}}(-)) = \overleftarrow{\pi} \circ \llbracket - \rrbracket$ by the universal property of \mathbf{Syn} established in Corollary 15. □

12.4 Correctness result

We are now ready to establish the fundamental correctness result for both forward-mode and reverse-mode CHAD. Specifically, we prove that these techniques yield the correct derivatives for any well-typed program of the form $x : \tau \vdash t : \sigma$, where τ and σ are types constructed from sum and product types.

Theorem 124 (Correctness of CHAD for tuples and variant tuples). *Let $(n_j)_{j \in J}$, $(m_l)_{l \in L}$, $((q_{(j,i)})_{i \in \{1, \dots, n_j\}})_{j \in J}$ and $((s_{(l,t)})_{t \in \{1, \dots, m_l\}})_{l \in L}$ be finite families of natural numbers.*

For any well-typed program $x : \tau \vdash t : \sigma$, where

$$\tau = \coprod_{j \in J} \left(\prod_{i=1}^{n_j} \mathbf{real}^{q_{(j,i)}} \right) \quad \text{and} \quad \sigma = \coprod_{l \in L} \left(\prod_{t=1}^{m_l} \mathbf{real}^{s_{(l,t)}} \right), \tag{155}$$

we have that $\llbracket t \rrbracket$ is differentiable. Moreover, (156) and (157) hold

$$\Sigma \llbracket \overrightarrow{\mathcal{D}}(t) \rrbracket = \mathcal{D} \llbracket t \rrbracket \tag{156} \qquad \overset{t}{\Sigma} \llbracket \overleftarrow{\mathcal{D}}(t) \rrbracket = \mathcal{D}^t \llbracket t \rrbracket \tag{157}$$

Proof. Let $t : \coprod_{j \in J} \left(\prod_{i=1}^{n_j} \mathbf{real}^{q_{(j,i)}} \right) \rightarrow \coprod_{l \in L} \left(\prod_{t=1}^{m_l} \mathbf{real}^{s_{(l,t)}} \right)$ be a morphism in \mathbf{Syn} . By the commutativity of Diagram 152, the morphism $(\llbracket t \rrbracket, \Sigma \llbracket \overrightarrow{\mathcal{D}}(t) \rrbracket, \overset{t}{\Sigma} \llbracket \overleftarrow{\mathcal{D}}(t) \rrbracket)$ in $\mathbf{Fam}(\mathbf{Set}) \times \mathbf{Fam}(\mathbf{Vect}) \times \mathbf{Fam}(\mathbf{Vect}^{\text{op}})$ satisfies $\overleftarrow{\pi} \left(\overleftarrow{\llbracket t \rrbracket} \right) = (\llbracket t \rrbracket, \Sigma \llbracket \overrightarrow{\mathcal{D}}(t) \rrbracket, \overset{t}{\Sigma} \llbracket \overleftarrow{\mathcal{D}}(t) \rrbracket)$.

By Theorem 121, we conclude that $\llbracket t \rrbracket$ is differentiable and $\left(\Sigma \llbracket \overrightarrow{\mathcal{D}}(t) \rrbracket, {}^t \Sigma \llbracket \overleftarrow{\mathcal{D}}(t) \rrbracket \right) = (\mathcal{D} \llbracket t \rrbracket, \mathcal{D}^t \llbracket t \rrbracket)$. □

13. Inductive Data Types: μ -Polynomials

We establish the correctness of CHAD for any well-typed program of the form:

$$x : \tau \vdash t : \sigma,$$

where τ and σ are data types in our source language in Section 13.3.

It should be noted that our source language supports inductive types, which enable us to represent lists, trees, or other more complex inductive types. In order to emphasize this fact, we refer to our data types as *inductive data types*.

We start by clarifying the categorical semantics of inductive data types, which are referred to as μ -polynomials and defined in Section 13.1. We demonstrate in Section 13.2 how μ -polynomials can be created from coproducts and finite products in concrete models that feature infinite coproducts. As a result, we can deduce that whenever τ is an inductive data type, $\llbracket \tau \rrbracket$ is an Euclidean family. This allows us to establish the specification and correctness of forward- and reverse-mode CHAD for general inductive data types in Section 13.3.

The definitions and results presented below heavily rely on the terminology, notation, and results established in Sections 3.6 and 4.

13.1 μ -polynomials

In our source language, data types are constructed using tupling, cotupling, and the μ -fixpoint operator. From a categorical semantic viewpoint, this implies that we want to examine objects that arise from products, coproducts, and initial algebras. Specifically, we consider the μ -polynomials as defined below.

Definition 125 (μ -polynomials). *The set μPoly of μ -polynomial functors in \mathbf{Syn} is the smallest set satisfying $(\mu\text{Poly}1)$, $(\mu\text{Poly}2)$, $(\mu\text{Poly}3)$, $(\mu\text{Poly}4)$, and $(\mu\text{Poly}5)$.*

$(\mu\text{Poly}1)$ For every $k \in \mathbb{N}$, every projection $\pi_t : \mathbf{Syn}^k \rightarrow \mathbf{Syn}$ is an element μPoly .

$(\mu\text{Poly}2)$ For any $k \in \mathbb{N}$, the constant functors:

$$\mathbb{1} : \mathbf{Syn}^k \rightarrow \mathbf{Syn}, W \mapsto \mathbb{1} \quad \text{and} \quad \mathbb{0} : \mathbf{Syn}^k \rightarrow \mathbf{Syn}, W \mapsto \mathbb{0}$$

belong to μPoly .

$(\mu\text{Poly}3)$ For any $k \in \mathbb{N}$ and any primitive type $\mathbf{real}^n \in \text{obj}(\mathbf{Syn})$, the functor:

$$H_{\mathbf{real}^n} : \mathbf{Syn}^k \rightarrow \mathbf{Syn}$$

constantly equal to \mathbf{real}^n belongs to μPoly .

$(\mu\text{Poly}4)$ If $H : \mathbf{Syn}^k \rightarrow \mathbf{Syn}$ and $J : \mathbf{Syn}^k \rightarrow \mathbf{Syn}$ are functors in μPoly , then (158) and (159) belong to μPoly :

$$\times \circ (H, J) : \mathbf{Syn}^k \rightarrow \mathbf{Syn}, W \mapsto H(W) \times J(W) \tag{158}$$

$$\sqcup \circ (H, J) : \mathbf{Syn}^k \rightarrow \mathbf{Syn}, W \mapsto H(W) \sqcup J(W) \tag{159}$$

$(\mu\text{Poly}5)$ If $k \in \mathbb{N} - \{0\}$ and $H : \mathbf{Syn}^k \rightarrow \mathbf{Syn}$ belongs to μPoly , then the parameterized initial algebra (initial algebra) $\mu H : \mathbf{Syn}^{k-1} \rightarrow \mathbf{Syn}(\mu H)$ belongs to μPoly .

An inductive data type is a type τ in our source language that corresponds to a initial algebra of a μ -polynomial functor $E : \mathbf{Syn} \rightarrow \mathbf{Syn}$.

13.2 μ -polynomials in concrete models: a normal form

Similarly to Euclidean families, in concrete models of our source language, we can reduce every μ -polynomial functor to a canonically isomorphic normal form. More precisely, we have Theorem 126.

Let $G : \mathbf{Syn} \rightarrow \mathcal{D}$ be a strictly cartesian closed functor that strictly preserves $\mu\nu$ -polynomials. Given functors $H : \mathbf{Syn}^k \rightarrow \mathbf{Syn}$ and $J : \mathcal{D}^n \rightarrow \mathcal{D}$, we say that J is (H, G) -compatible if (160) commutes

$$\begin{array}{ccc}
 \mathbf{Syn}^k & \xrightarrow{G^k} & \mathcal{D}^k \\
 H \downarrow & & \downarrow J \\
 \mathbf{Syn} & \xrightarrow{G} & \mu\mathcal{D}
 \end{array} \tag{160}$$

In the result below, we denote $\mathbb{I}_n := \{1, \dots, n\}$, for each $n \in \mathbb{N}$.

Theorem 126. *Let \mathcal{D} be a cartesian closed category with $\mu\nu$ -polynomials and infinite coproducts. We assume that $G : \mathbf{Syn} \rightarrow \mathcal{D}$ is strictly cartesian closed functor that strictly preserves $\mu\nu$ -polynomials.*

If $H : \mathbf{Syn}^n \rightarrow \mathbf{Syn}$ is a functor in μPoly , then there is a quadruple $(J, \mathfrak{N}H, m, \mathfrak{n})$, where $F : \mathcal{D}^n \rightarrow \mathcal{D}$ is an (H, G) -compatible functor, $m = \left(m_{(j,T)}\right)_{(j,T) \in (\mathbb{I}_n \cup \{0\}) \times \mathbf{T}}$ is a countable family of natural numbers and

$$\mathfrak{n}_{(Y_i)_{i \in \mathbb{I}_n}} : F(Y_i)_{i \in \mathbb{I}_n} \cong \coprod_{T \in \mathbf{T}} \left(\mathfrak{N}_T^{m(0,T)} \times \prod_{j=1}^n Y_j^{m(j,T)} \right) \tag{161}$$

is a natural isomorphism, where, for each $T \in \mathbf{T}$,

$$\mathfrak{N}_T = \prod_{l \in L_T} G(\mathbf{real}^{z(l,T)}) \tag{162}$$

for some finite family $(z_{(l,T)})_{l \in L_T}$ of natural numbers.

Proof. The result follows from induction over the definition of μPoly . The only nontrivial part of the proof is related to ($\mu\text{Poly}5$), that is to say, the stability of μPoly under the parameterized initial algebras, which we sketch below.

Let $\tilde{H} : \mathbf{Syn}^{n+1} \rightarrow \mathbf{Syn}$ be a member of μPoly . We assume, by induction, that $\tilde{F} : \mathcal{D}^{n+1} \rightarrow \mathcal{D}$ satisfies the above. That is to say, it is an (H, G) -compatible functor and we have a natural isomorphism:

$$\tilde{F}(Y_i)_{i \in \mathbb{I}_{n+1}} \cong \coprod_{r \in \mathcal{L}} \left(\tilde{\mathfrak{N}}_r^{\mathfrak{s}(0,r)} \times \prod_{i=1}^{n+1} Y_i^{\mathfrak{s}(i,r)} \right).$$

where $\tilde{\mathfrak{N}}_r$ is equal to some finite product:

$$\prod_{l \in L_r} G(\mathbf{real}^{z(l,r)}).$$

It is clear that \tilde{F} preserves colimits of ω -chains. Hence, given $W = (W_i)_{i \in \mathbb{I}_n}$, $\mu\tilde{F}^W = \mu\tilde{F}(W)$ exists and is given by the colimit of the ω -chain:

$$0 \rightarrow \tilde{F}^W(0) \rightarrow (\tilde{F}^W)^2(0) \rightarrow \dots \tag{163}$$

provided that it exists.

We claim that the colimit (163) indeed exists. More precisely, the colimit is given by the coproduct:

$$\coprod_{q=0}^{\infty} S_q(W)$$

where $(S_q(W))_{q \in \mathbb{N}}$ is defined inductively by (S1) and (S2).

(S1) Denoting by $\bar{K}_0 := \{r \in \mathcal{L} \text{ such that } s_{(n+1,r)} = 0\}$,

$$S_0(W) := \coprod_{r \in \bar{K}_0} \left(\tilde{\mathfrak{N}}_r \times \prod_{i=1}^n W_i^{s(i,r)} \right).$$

(S2) Denoting by $\bar{K}_a := \{r \in \mathcal{L} \text{ such that } s_{(n+1,r)} = a\}$,

$$S_{q+1}(W) := \coprod_{a=1}^{\infty} \coprod_{r \in \bar{K}_a} \left((S_q(W))^a \times \tilde{\mathfrak{N}}_r \times \prod_{i=1}^n W_i^{s(i,r)} \right).$$

By the infinitely distributive property and the universal property of the coproduct and product, we conclude that there is a canonical isomorphism between

$$\mu\tilde{F}(W) = \coprod_{q=0}^{\infty} S_q(W)$$

and something of the form $\coprod_{T \in \mathbf{T}} \left(\mathfrak{N}_T^{m(0,T)} \times \prod_{j=1}^n Y_j^{m(j,T)} \right)$, as described in (161).

Since G preserves $\mu\nu$ -polynomials, we conclude that $\mu\tilde{F}$ is a $(\mu\tilde{H}, G)$ -compatible satisfying the required conditions. □

As consequence, we get:

Corollary 127. *Let \mathcal{D} be a cartesian closed category with $\mu\nu$ -polynomials and infinite coproducts. We assume that $G: \mathbf{Syn} \rightarrow \mathcal{D}$ is strictly cartesian closed functor that strictly preserves $\mu\nu$ -polynomials. If $E: \mathbf{Syn} \rightarrow \mathbf{Syn}$ is an endofunctor in $\mu\mathbf{Poly}$, then there is a canonical isomorphism:*

$$\mathfrak{N}: G(\mu E) \cong \coprod_{l \in L} \left(\prod_{t=1}^{m_l} G(\mathbf{real}^{s(l,t)}) \right), \tag{164}$$

where $(m_l)_{l \in L}$ and $\left((s(l,t))_{t \in \{1, \dots, m_l\}} \right)_{l \in L}$ are (possibly infinite) families of natural numbers.

13.3 Correctness of CHAD for inductive data types, by logical relations

Since the canonical isomorphisms \mathfrak{N} given in Corollary 127 are indeed canonical in the sense that they are given by the composition of isomorphisms coming from the distributivity property and universal property of (co)products, we have that:

Lemma 128. *Let τ be an inductive data type as defined in Section 13.1. It follows that there is a canonical isomorphism:*

$$\mathfrak{N}_\tau : \overleftarrow{\llbracket \tau \rrbracket} \cong \coprod_{l \in L} \left(\prod_{t=1}^{m_l} \overleftarrow{\llbracket \mathbf{real}^{s(l,t)} \rrbracket} \right), \tag{165}$$

such that:

- (C1) $(m_l)_{l \in L}$ and $\left((s(l,t))_{t \in \{1, \dots, m_l\}} \right)_{l \in L}$ are (possibly infinite) families of natural numbers;
- (C2) $\underline{\mathfrak{N}}_\tau$ is a diffeomorphism;
- (C3) $\mathfrak{N}_\tau = (\underline{\mathfrak{N}}_\tau, \mathfrak{D}(\underline{\mathfrak{N}}_\tau), \mathfrak{D}^t(\underline{\mathfrak{N}}_\tau))$.

By making use of the canonical isomorphisms (165), we can prove our correctness theorem; namely:

Theorem 129 (Correctness of CHAD for tuples and variant tuples). *For any well-typed program $x : \tau \vdash t : \sigma$, where τ, σ are inductive data types, we have that $\llbracket t \rrbracket$ is differentiable. Moreover, (166) and (167) hold*

$$\Sigma \llbracket \vec{\mathcal{D}}[t]() \rrbracket = \mathfrak{D} \llbracket t \rrbracket \tag{166} \qquad \Sigma^t \llbracket \overleftarrow{\mathcal{D}}[t]() \rrbracket = \mathfrak{D}^t \llbracket t \rrbracket \tag{167}$$

Proof. Let $t : \tau \rightarrow \sigma$ be a morphism in **Syn**. By the commutativity of Diagram 152 and Lemma 128, the morphism $\left(\llbracket t \rrbracket, \Sigma \llbracket \vec{\mathcal{D}}[t]() \rrbracket, \Sigma^t \llbracket \overleftarrow{\mathcal{D}}[t]() \rrbracket \right)$ in $\mathbf{Fam}(\mathbf{Set}) \times \mathbf{Fam}(\mathbf{Vect}) \times \mathbf{Fam}(\mathbf{Vect}^{\text{op}})$ is such that $\overleftarrow{\mathfrak{N}} \left(\mathfrak{N}_\sigma \circ \overleftarrow{\llbracket t \rrbracket} \circ \mathfrak{N}_\tau^{-1} \right)$ is equal to

$$\left(\underline{\mathfrak{N}}_\sigma \circ \llbracket t \rrbracket \circ \underline{\mathfrak{N}}_\tau^{-1}, \mathfrak{D}(\underline{\mathfrak{N}}_\sigma) \circ \Sigma \llbracket \vec{\mathcal{D}}[t]() \rrbracket \circ \mathfrak{D}(\underline{\mathfrak{N}}_\tau)^{-1}, \mathfrak{D}^t(\underline{\mathfrak{N}}_\sigma) \circ \Sigma^t \llbracket \overleftarrow{\mathcal{D}}[t]() \rrbracket \circ \mathfrak{D}^t(\underline{\mathfrak{N}}_\tau)^{-1} \right).$$

By Theorem 121, we conclude that

- (C1) $\underline{\mathfrak{N}}_\sigma \circ \llbracket t \rrbracket \circ \underline{\mathfrak{N}}_\tau^{-1}$ is differentiable;
- (C2) $\left(\mathfrak{D}(\underline{\mathfrak{N}}_\sigma) \circ \Sigma \llbracket \vec{\mathcal{D}}[t]() \rrbracket \circ \mathfrak{D}(\underline{\mathfrak{N}}_\tau)^{-1}, \mathfrak{D}^t(\underline{\mathfrak{N}}_\sigma) \circ \Sigma^t \llbracket \overleftarrow{\mathcal{D}}[t]() \rrbracket \circ \mathfrak{D}^t(\underline{\mathfrak{N}}_\tau)^{-1} \right) = (\mathfrak{D} \llbracket t \rrbracket, \mathfrak{D}^t \llbracket t \rrbracket)$.

By the chain rule, since $\underline{\mathfrak{N}}_\sigma$ and $\underline{\mathfrak{N}}_\tau$ are diffeomorphisms, we conclude that $\llbracket t \rrbracket$ is differentiable and $\left(\Sigma \llbracket \vec{\mathcal{D}}[t]() \rrbracket, \Sigma^t \llbracket \overleftarrow{\mathcal{D}}[t]() \rrbracket \right) = (\mathfrak{D} \llbracket t \rrbracket, \mathfrak{D}^t \llbracket t \rrbracket)$. □

14. Examples of Reverse-Mode CHAD

We provide examples of reverse-mode CHAD computation of derivatives, with a focus on computing derivatives of functions involving inductive types. In particular, we consider the simplest example of an inductive type: the type of nonempty lists of real numbers, denoted by \mathbf{real}_* .

We present three examples. The function $\text{sum} : [\mathbf{real}]_* \rightarrow \mathbf{real}$ that computes the sum of elements of a list; $\text{product} : [\mathbf{real}]_* \rightarrow \mathbf{real}$ that gives the product of elements of a list; and the polynomial evaluator $\text{ev}_{poly} : [\mathbf{real}]_* \rightarrow \mathbf{real}$. The semantics of these functions are roughly described below:

$$\llbracket \text{sum} \rrbracket : \llbracket [\mathbf{real}]_* \rrbracket \rightarrow \llbracket \mathbf{real} \rrbracket, \quad [a_0, \dots, a_n] \mapsto a_0 + a_1 + \dots + a_n \quad (168)$$

$$\llbracket \text{product} \rrbracket : \llbracket [\mathbf{real}]_* \rrbracket \rightarrow \llbracket \mathbf{real} \rrbracket, \quad [a_0, \dots, a_n] \mapsto a_0 a_1 \dots a_n \quad (169)$$

$$\llbracket \text{ev}_{poly} \rrbracket : \llbracket [\mathbf{real}]_* \rrbracket \rightarrow \llbracket \mathbf{real} \rrbracket, \quad [a_0, \dots, a_n, v] \mapsto a_0 + a_1 v + \dots + a_n v^n \quad (170)$$

The examples presented below heavily rely on the terminology, notation, and results established in Sections 3.6, 4, and 13.

14.1 The derivative of 0

In order to express the polynomial evaluator, we assume that we have a morphism $0 : \mathbf{real} \rightarrow \mathbf{real}$ whose semantics correspond to the function $0 : \mathbb{R} \rightarrow \mathbb{R}$ constantly equal to $0 \in \mathbb{R}$.

The morphism $0 : \mathbf{real} \rightarrow \mathbf{real}$ can be either a primitive operation, or a function obtained by composing

$$\mathbf{real} \rightarrow \mathbb{1} \xrightarrow{0} \mathbf{real},$$

where the constant $0 : \mathbb{1} \rightarrow \mathbf{real}$ would be taken to be the primitive operation. Either way, by our semantic assumptions of Section 10.6, we get that

$$\overset{t}{\Sigma} \llbracket \overleftarrow{\mathcal{D}}(0) \rrbracket : (\mathbb{R}, \underline{\mathbb{R}}) \rightarrow (\mathbb{R}, \underline{\mathbb{R}}) \quad (171)$$

is the morphism in $\mathbf{Fam}(\mathbf{Vect}^{\text{op}})$ defined by the pair $(0, 0')$ where, for each $a \in \mathbb{R}$, $0'_a : \mathbb{R} \rightarrow \mathbb{R}$ is the linear transformation constantly equal to 0.

14.2 The derivatives of (+) and (·)

We assume that

$$(\cdot) : \mathbf{real} \times \mathbf{real} \rightarrow \mathbf{real} \quad \text{and} \quad (+) : \mathbf{real} \times \mathbf{real} \rightarrow \mathbf{real}$$

are primitive operations in the source language whose semantics are given, respectively, by the addition $\text{plus} : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$ and multiplication $\text{multi} : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$. Since $(+)$ and (\cdot) are primitive operations in the source language, $\overleftarrow{\mathcal{D}}(+)$ and $\overleftarrow{\mathcal{D}}(\cdot)$ are set by definition.

By our semantic assumptions as per Section 10.6, we have:

$(+)$ the morphism $\overset{t}{\Sigma} \llbracket \overleftarrow{\mathcal{D}}(+)\rrbracket : (\mathbb{R}, \underline{\mathbb{R}}) \rightarrow (\mathbb{R} \times \mathbb{R}, \underline{\mathbb{R}} \times \underline{\mathbb{R}})$ of $\mathbf{Fam}(\mathbf{Vect}^{\text{op}})$ is defined by:

$$\overset{t}{\Sigma} \llbracket \overleftarrow{\mathcal{D}}(+)\rrbracket = (\text{plus}, \text{plus}') \quad (172)$$

where $\text{plus}(a, b) = a + b$ and, for each $(a, b) \in \mathbb{R} \times \mathbb{R}$, $\text{plus}'_{(a,b)} : \mathbb{R} \rightarrow \mathbb{R} \times \mathbb{R}$ is defined by $x \mapsto (x, x)$.

(\cdot) the morphism $\overset{t}{\Sigma} \llbracket \overleftarrow{\mathcal{D}}(\cdot)\rrbracket : (\mathbb{R}, \underline{\mathbb{R}}) \rightarrow (\mathbb{R} \times \mathbb{R}, \underline{\mathbb{R}} \times \underline{\mathbb{R}})$ of $\mathbf{Fam}(\mathbf{Vect}^{\text{op}})$ is defined by:

$$\overset{t}{\Sigma} \llbracket \overleftarrow{\mathcal{D}}(\cdot)\rrbracket = (\text{multi}, \text{multi}') \quad (173)$$

where $\text{multi}(a, b) = ab$ and, for each $(a, b) \in \mathbb{R} \times \mathbb{R}$, $\text{multi}'_{(a,b)} : \mathbb{R} \rightarrow \mathbb{R} \times \mathbb{R}$ is defined by $x \mapsto (bx, ax)$.

14.3 Type of nonempty lists of real numbers in \mathbf{Syn} and $\mathbf{Fam}(\mathbf{Vect}^{\text{op}})$

As our examples mainly concern the type $[\mathbf{real}]_*$ of nonempty lists of real numbers in \mathbf{Syn} , let us first recall its categorical semantics and discuss its image under the reverse-mode CHAD $\overleftarrow{\mathcal{D}}(-)$.

The $[\mathbf{real}]_* := \mu E$ where the endofunctor E is defined by:

$$E : \mathbf{Syn} \rightarrow \mathbf{Syn} \tag{174}$$

$$W \mapsto \mathbf{real} \sqcup W \times \mathbf{real}.$$

Denoting by $\mathcal{E} : \mathbf{Fam}(\mathbf{Vect}^{\text{op}}) \rightarrow \mathbf{Fam}(\mathbf{Vect}^{\text{op}})$ the endofunctor defined by:

$$\mathcal{E}(W, w) = (\mathbb{R}, \underline{\mathbb{R}}) \sqcup (W, w) \times (\mathbb{R}, \underline{\mathbb{R}}), \tag{175}$$

we conclude that

$$\begin{aligned} \overset{t}{\Sigma} [\overleftarrow{\mathcal{D}}([\mathbf{real}]_*)] &= \mu \mathcal{E} \tag{176} \\ &= \left(\coprod_{j \in \mathbb{N} - \{0\}} \mathbb{R}^j, \langle \underline{\mathbb{R}}^j \rangle_{j \in \mathbb{N} - \{0\}} : \coprod_{j \in \mathbb{N} - \{0\}} \mathbb{R}^j \rightarrow \mathbf{Vect} \right) \end{aligned}$$

by the structure-preserving property of $\overleftarrow{\mathcal{D}}(-)$.

Let $\langle (\zeta, \zeta'), (\beta, \beta') \rangle : (\mathbb{R}, \underline{\mathbb{R}}) \sqcup (W, w) \times (\mathbb{R}, \underline{\mathbb{R}}) \rightarrow (W, w)$ be the morphism in $\mathbf{Fam}(\mathbf{Vect}^{\text{op}})$ induced by given morphisms:

$$\begin{aligned} (\zeta, \zeta') : (\mathbb{R}, \underline{\mathbb{R}}) &\rightarrow (W, w) \\ (\beta, \beta') : (W, w) \times (\mathbb{R}, \underline{\mathbb{R}}) &\rightarrow (W, w) \end{aligned}$$

in $\mathbf{Fam}(\mathbf{Vect}^{\text{op}})$. Denoting

$$(\xi, \xi') := \text{fold}_{\mathcal{E}}((W, w), \langle (\zeta, \zeta'), (\beta, \beta') \rangle) : \mu \mathcal{E} \rightarrow (W, w), \tag{177}$$

we have the following:

(ξ) $\xi : \coprod_{j \in \mathbb{N} - \{0\}} \mathbb{R}^j \rightarrow W$ is induced by the family:

$$\xi = \langle \xi_j : \mathbb{R}^j \rightarrow W \rangle_{j \in \mathbb{N} - \{0\}} \tag{178}$$

defined by $\xi_1 = \zeta : \mathbb{R} \rightarrow W$ and $\xi_{j+1} = \beta \circ (\xi_j \times \text{id}_{\mathbb{R}})$;

(ξ'_r) for each $r \in \mathbb{R} \subset \coprod_{j \in \mathbb{N} - \{0\}} \mathbb{R}^j$, the component:

$$\xi'_r : w \circ \xi(r) \rightarrow \mathbb{R}$$

is given by $\xi'_r : w \circ \zeta(r) \rightarrow \mathbb{R}$.

(ξ'_p) for each $p = (p_*, p_0) \in \mathbb{R}^k \times \mathbb{R} = \mathbb{R}^{k+1} \subset \coprod_{j \in \mathbb{N} - \{0\}} \mathbb{R}^j$,

$$\xi'_p = \left(\xi'_{p_*} \times \text{id}_{\mathbb{R}} \right) \circ \beta'_{(\xi(p_*), p_0)}. \tag{179}$$

14.4 Reverse-mode CHAD derivative of sum

The function $\text{sum} : [\mathbf{real}]_* \rightarrow \mathbf{real}$ computes the sum of the elements of a nonempty list of real numbers. We can express sum in \mathbf{Syn} by:

$$\text{sum} := \text{fold}_E(\mathbf{real}, \langle \text{id}_{\mathbf{real}}, (+) \rangle : \mathbf{real} \sqcup \mathbf{real} \times \mathbf{real} \rightarrow \mathbf{real}) : \mu E = [\mathbf{real}]_* \rightarrow \mathbf{real}. \tag{180}$$

By the structure-preserving property of CHAD, we conclude that

$${}^t_{\Sigma} \llbracket \overleftarrow{\mathcal{D}}(\text{sum}) \rrbracket = \text{fold}_{\mathcal{E}} \left((\mathbb{R}, \underline{\mathbb{R}}), \langle \text{id}_{(\mathbb{R}, \underline{\mathbb{R}})}, (\text{plus}, \text{plus}') \rangle : (\mathbb{R}, \underline{\mathbb{R}}) \sqcup (\mathbb{R} \times \mathbb{R}, \underline{\mathbb{R}}^2) \rightarrow (\mathbb{R}, \underline{\mathbb{R}}) \right). \tag{181}$$

Therefore, by (14.2) and (14.3), we conclude that, denoting ${}^t_{\Sigma} \llbracket \overleftarrow{\mathcal{D}}(\text{sum}) \rrbracket = (\llbracket \text{sum} \rrbracket, \llbracket \text{sum}' \rrbracket)$, we have the following:

(A) the function

$$\llbracket \text{sum} \rrbracket : \coprod_{j \in \mathbb{N} - \{0\}} \mathbb{R}^j \rightarrow \mathbb{R} \tag{182}$$

is induced by the family $\langle \llbracket \text{sum} \rrbracket_j : \mathbb{R}^j \rightarrow \mathbb{R} \rangle_{j \in \mathbb{N} - \{0\}}$ defined by:

$$\llbracket \text{sum} \rrbracket_j (w_1, \dots, w_j) = \sum_{i=1}^j w_i;$$

(B) for each $p \in \mathbb{R}^k \subset \coprod_{j \in \mathbb{N} - \{0\}} \mathbb{R}^j$, we have that

$$\llbracket \text{sum}' \rrbracket'_p : \mathbb{R} \rightarrow \mathbb{R}^k \tag{183}$$

is defined by $x \mapsto (x, \dots, x)$.

14.5 Reverse-mode CHAD derivative of product

The function product : $[\mathbf{real}]_* \rightarrow \mathbf{real}$ computes the product of the elements of a nonempty list of real numbers. We can express product in **Syn** by:

$$\text{product} := \text{fold}_{\mathcal{E}} (\mathbf{real}, \langle \text{id}_{\mathbf{real}}, (\cdot) \rangle : \mathbf{real} \sqcup \mathbf{real} \times \mathbf{real} \rightarrow \mathbf{real}) : \mu E = [\mathbf{real}]_* \rightarrow \mathbf{real}. \tag{184}$$

By the structure-preserving property of CHAD, we have that

$${}^t_{\Sigma} \llbracket \overleftarrow{\mathcal{D}}(\text{sum}) \rrbracket = \text{fold}_{\mathcal{E}} \left((\mathbb{R}, \underline{\mathbb{R}}), \langle \text{id}_{(\mathbb{R}, \underline{\mathbb{R}})}, (\text{multi}, \text{multi}') \rangle : (\mathbb{R}, \underline{\mathbb{R}}) \sqcup (\mathbb{R} \times \mathbb{R}, \underline{\mathbb{R}}^2) \rightarrow (\mathbb{R}, \underline{\mathbb{R}}) \right). \tag{185}$$

Therefore, by (14.2) and (14.3), we conclude that, denoting ${}^t_{\Sigma} \llbracket \overleftarrow{\mathcal{D}}(\text{product}) \rrbracket = (\llbracket \text{product} \rrbracket, \llbracket \text{product}' \rrbracket)$, we have the following:

(I) the function

$$\llbracket \text{product} \rrbracket : \coprod_{j \in \mathbb{N} - \{0\}} \mathbb{R}^j \rightarrow \mathbb{R} \tag{186}$$

is induced by the family $\langle \llbracket \text{product} \rrbracket_j : \mathbb{R}^j \rightarrow \mathbb{R} \rangle_{j \in \mathbb{N} - \{0\}}$ defined by:

$$\llbracket \text{product} \rrbracket_j (w_1, \dots, w_j) = \prod_{i=1}^j w_i;$$

(II) for each

$$p = (p_1, \dots, p_k) \in \mathbb{R}^k \subset \coprod_{j \in \mathbb{N} - \{0\}} \mathbb{R}^j,$$

we have that

$$\llbracket \text{product}' \rrbracket'_p : \mathbb{R} \rightarrow \mathbb{R}^k \tag{187}$$

is defined by $x \mapsto (\hat{p}_1x, \hat{p}_2x, \dots, \hat{p}_kx)$, where

$$\hat{p}_t = \prod_{i \in \{1, \dots, k\} - \{t\}} p_i.$$

14.6 Reverse-mode CHAD derivative of $(+) \circ (\text{id}_{\text{real}} \times (\cdot))$

In order to compute the derivative of the polynomial evaluator as expressed in (192), we need to compute the derivative of the function:

$$(\cdot) \circ (\text{id}_{\text{real}} \times (\cdot)) : \text{real} \times \text{real} \times \text{real} \rightarrow \text{real} \tag{188}$$

whose semantics is defined by $(a, b, c) \mapsto a + bc$.

We use the structure-preserving property of $\overleftarrow{\mathcal{D}}(-)$ to compute $\overset{t}{\Sigma} \llbracket \overleftarrow{\mathcal{D}}(\text{id}_{\text{real}} \times (\cdot)) \rrbracket$. This gives us:

$$\begin{aligned} \overset{t}{\Sigma} \llbracket \overleftarrow{\mathcal{D}}(\text{id}_{\text{real}} \times (\cdot)) \rrbracket &= \overset{t}{\Sigma} \llbracket \overleftarrow{\mathcal{D}}(\text{id}_{\text{real}}) \rrbracket \times \overset{t}{\Sigma} \llbracket \overleftarrow{\mathcal{D}}(\cdot) \rrbracket \\ &= \text{id}_{(\mathbb{R}, \mathbb{R})} \times (\text{multi}, \text{multi}') \\ &= (\overline{\text{multi}}, \overline{\text{multi}}') \end{aligned}$$

in $\mathbf{Fam}(\mathbf{Vect}^{\text{op}})$, where $\overline{\text{multi}} : \mathbb{R} \times \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R} \times \mathbb{R}$ is defined by $(a, b, c) \mapsto (a, bc)$ and, for each $(a, b, c) \in \mathbb{R} \times \mathbb{R} \times \mathbb{R}$,

$$\overline{\text{multi}}'_{(a,b,c)} : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R} \times \mathbb{R} \times \mathbb{R}, \quad (w, x) \mapsto (w, cx, bx). \tag{189}$$

We conclude, then, that

$$\begin{aligned} \overset{t}{\Sigma} \llbracket \overleftarrow{\mathcal{D}}((+) \circ (\text{id}_{\text{real}} \times (\cdot))) \rrbracket &= \overset{t}{\Sigma} \llbracket \overleftarrow{\mathcal{D}}(+) \rrbracket \circ \overset{t}{\Sigma} \llbracket \overleftarrow{\mathcal{D}}(\text{id}_{\text{real}} \times (\cdot)) \rrbracket \\ &= \overset{t}{\Sigma} \llbracket \overleftarrow{\mathcal{D}}(+) \rrbracket \circ \left(\overset{t}{\Sigma} \llbracket \overleftarrow{\mathcal{D}}(\text{id}_{\text{real}}) \rrbracket \times \overset{t}{\Sigma} \llbracket \overleftarrow{\mathcal{D}}(\cdot) \rrbracket \right) \\ &= (\text{plus}, \text{plus}') \circ (\overline{\text{multi}}, \overline{\text{multi}}') \end{aligned}$$

is equal to the morphism:

$$(\overline{\text{plus}}, \overline{\text{plus}})' : - (\text{plus} \circ (\text{id}_{\mathbb{R}} \times \text{multi}), (\text{plus} \circ (\text{id}_{\mathbb{R}} \times \text{multi})))' : (\mathbb{R} \times \mathbb{R} \times \mathbb{R}, \mathbb{R}^3) \rightarrow (\mathbb{R}, \mathbb{R}) \tag{190}$$

where, for each $(a, b, c) \in \mathbb{R} \times \mathbb{R} \times \mathbb{R}$,

$$\begin{aligned} \overline{\text{plus}}'_{(a,b,c)} &= (\text{plus} \circ (\text{id}_{\mathbb{R}} \times \text{multi}))'_{(a,b,c)} : \mathbb{R} \rightarrow \mathbb{R} \times \mathbb{R} \times \mathbb{R} \\ &x \mapsto (x, cx, bx). \end{aligned}$$

14.7 Reverse-mode CHAD derivative of polynomial evaluator

For convenience, we represent a pair $(p(x), v)$, where

$$p(x) = a_0 + \dots + a_nx^n \tag{191}$$

is a polynomial and $v \in \mathbb{R}$, by a nonempty list $[a_0, \dots, a_n, v]$. With this notation, the polynomial evaluator:

$$\text{ev}_{\text{poly}} : [\text{real}]_* \rightarrow \text{real}$$

can be expressed as the composition:

$$\mu E = [\text{real}]_* \xrightarrow{\text{fold}_E(\text{real} \times \text{real}, ((0, \text{id}_{\text{real}}), ((+) \circ (\text{id}_{\text{real}} \times (\cdot)), \pi_3)))} \text{real} \times \text{real} \xrightarrow{\pi_1} \text{real}. \tag{192}$$

It should be noted that $\llbracket \langle (0, \text{id}_{\mathbb{R}}), ((+) \circ (\text{id}_{\mathbb{R}} \times (\cdot)), \pi_3) \rangle \rrbracket$ is the morphism:

$$\mathbb{R} \sqcup (\mathbb{R} \times \mathbb{R} \times \mathbb{R}) \rightarrow \mathbb{R} \times \mathbb{R}$$

in **Fam(Set)** induced by the morphism $\mathbb{R} \ni r \mapsto (0, r)$ and $\mathbb{R} \times \mathbb{R} \times \mathbb{R} \ni (a, b, c) \mapsto (c, a + bc)$ and, hence, indeed,

$$\llbracket \text{ev}_{poly} \rrbracket (a_0, \dots, a_k, v) = (a_0 + \dots + a_k v^k, v)$$

for each $(a_0, \dots, a_k, v) \in \mathbb{R}^k \subset \coprod_{j \in \mathbb{N} - \{0\}} \mathbb{R}^j$.

By the structure-preserving property of $\overleftarrow{\mathcal{D}}(-)$, we conclude that $\overset{t}{\Sigma} \llbracket \overleftarrow{\mathcal{D}}(\text{ev}_{poly}) \rrbracket$ is given by the composition:

$$\mu_{\mathcal{E}} \xrightarrow{\text{fold}_{\mathcal{E}}(\mathbb{R} \times \mathbb{R}, \langle (0, 0'), (\text{id}_{\mathbb{R}}, \text{id}'_{\mathbb{R}}) \rangle, \langle (\overline{\text{plus}}, \overline{\text{plus}'}) \rangle, \langle (\pi_3, \pi'_3) \rangle)} (\mathbb{R}^2, \underline{\mathbb{R}}^2) \xrightarrow{(\pi_1, \pi'_1)} (\mathbb{R}, \underline{\mathbb{R}}), \tag{193}$$

where (π_3, π'_3) and (π_1, π'_1) denote the respective projections in **Fam(Vect^{op})**. By Section 14.3, denoting

$$(g, g') := \text{fold}_{\mathcal{E}}(\mathbb{R} \times \mathbb{R}, \langle (0, 0'), (\text{id}_{\mathbb{R}}, \text{id}'_{\mathbb{R}}) \rangle, \langle (\overline{\text{plus}}, \overline{\text{plus}'}) \rangle, \langle (\pi_3, \pi'_3) \rangle), \tag{194}$$

we have the following.

(a) The function

$$g : \coprod_{j \in \mathbb{N} - \{0\}} \mathbb{R}^j \rightarrow \mathbb{R} \times \mathbb{R} \tag{195}$$

takes each

$$(a_0, \dots, a_k, v) \in \mathbb{R}^{k+1} \subset \coprod_{j \in \mathbb{N} - \{0\}} \mathbb{R}^j$$

to $(a_0 + a_1 v + \dots + a_k v^k, v) \in \mathbb{R} \times \mathbb{R}$.

(b) For each $(a_0, \dots, a_k, v) \in \mathbb{R}^{k+1} \subset \coprod_{j \in \mathbb{N} - \{0\}} \mathbb{R}^j$,

$$g'_{(a_0, \dots, a_k, v)} : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}^{k+1} \tag{196}$$

is defined by $(x, y) \mapsto (x, vx, v^2x, \dots, v^kx, (a_1 + 2 \cdot a_2v + 3 \cdot a_3v^2 + \dots + ka_kv^{k-1})x + y)$.

Therefore $(\llbracket \text{ev}_{poly} \rrbracket, \overset{t}{\Sigma} \llbracket \text{ev}_{poly} \rrbracket') := \overset{t}{\Sigma} \llbracket \overleftarrow{\mathcal{D}}(\text{ev}_{poly}) \rrbracket = (\pi_1, \pi'_1) \circ (g, g')$ is such that, for each

$$(a_0, \dots, a_k, v) \in \mathbb{R}^{k+1} \subset \coprod_{j \in \mathbb{N} - \{0\}} \mathbb{R}^j,$$

$\overset{t}{\Sigma} \llbracket \text{ev}_{poly} \rrbracket'_{(a_0, \dots, a_k, v)} : \mathbb{R} \rightarrow \mathbb{R}^{k+1}$ is defined by:

$$x \mapsto (x, vx, v^2x, \dots, v^kx, (a_1 + 2 \cdot a_2v + 3 \cdot a_3v^2 + \dots + ka_kv^{k-1})x).$$

15. Practical Considerations

Despite the theoretical approach this paper has taken, our motivations for this line of research are very applied: we want to achieve efficient and correct reverse AD on expressive programming languages. We believe this paper lays some of the necessary theoretical groundwork to achieve

that goal. We are planning to address the practical considerations around achieving efficient implementations of CHAD in detail in a dedicated applied follow-up paper. However, we still sketch some of these considerations in this section to convey that the methods described in this paper are not merely of theoretical interest.

15.1 Addressing expression blow-up and sharing common subcomputations

We can observe that our source code transformations of Appendix B can result in code blowup due to the interdependence of the transformations $\overrightarrow{\mathcal{D}}_{\overline{\Gamma}}(-)_1$ and $\overrightarrow{\mathcal{D}}_{\overline{\Gamma}}(-)_2$ (and $\overleftarrow{\mathcal{D}}_{\overline{\Gamma}-1}$ and $\overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(-)_2$, respectively) on programs. This is why, in Section 8, we have instead defined a single-code transformation on programs $\overrightarrow{\mathcal{D}}_{\overline{\Gamma}}(-)$ for forward mode and $\overleftarrow{\mathcal{D}}_{\overline{\Gamma}-}$ for reverse mode that simultaneously computes the primals and (co)tangents and shares any subcomputations they have in common. These more efficient CHAD transformations are still representations of the canonical CHAD functors $\overrightarrow{\mathcal{D}}(-) : \mathbf{Syn} \rightarrow \Sigma_{\mathbf{CSyn}} \mathbf{LSyn}$ and $\overleftarrow{\mathcal{D}}(-) : \mathbf{Syn} \rightarrow \Sigma_{\mathbf{CSyn}} \mathbf{LSyn}^{op}$ in the sense that $\overrightarrow{\mathcal{D}}_{\overline{\Gamma}}(t) \stackrel{\beta\eta+}{=} \langle \overrightarrow{\mathcal{D}}_{\overline{\Gamma}}(t)_1, \underline{\lambda v}. \overrightarrow{\mathcal{D}}_{\overline{\Gamma}}(t)_2 \rangle$ and $\overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(t) \stackrel{\beta\eta+}{=} \langle \overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(t)_1, \underline{\lambda v}. \overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(t)_2 \rangle$ and hence are equivalent to the inefficient CHAD transformations from the point of view of denotational semantics and correctness.

We can observe that the efficient CHAD code transformations $\overrightarrow{\mathcal{D}}_{\overline{\Gamma}}(-)$ and $\overleftarrow{\mathcal{D}}_{\overline{\Gamma}-}$ have the property that the transformation $\overrightarrow{\mathcal{D}}_{\overline{\Gamma}}(C[t_1, \dots, t_n])$ (resp. $\overleftarrow{\mathcal{D}}_{\overline{\Gamma}}C[t_1, \dots, t_n]$) of a term former $C[t_1, \dots, t_n]$ that takes n arguments t_1, \dots, t_n (e.g., the pair constructor $C[t_1, t_2] = \langle t_1, t_2 \rangle$, which takes two arguments t_1 and t_2) is a piece of code that uses the CHAD transformation $\overrightarrow{\mathcal{D}}_{\overline{\Gamma}}(t_i)$ (resp. $\overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(t_i)$) of each subterm t_i exactly once. This has as a consequence the following important compile-time complexity result that is a necessary condition if this AD technique is to scale up to large code bases.

Corollary 130 (No code blow-up). *The size of the code of the CHAD transformed programs $\overrightarrow{\mathcal{D}}_{\overline{\Gamma}}(t)$ and $\overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(t)$ grows linearly with the size of the original source program t .*

While we have taken care to avoid recomputation as much as possible in defining these code transformations by sharing results of subcomputations through **let**-bindings, the runtime complexity of the generated code remains to be studied.

15.2 Removing dependent types from the target language

In this paper, we have chosen to work with a dependently typed target language, as this allows our AD transformations to correspond as closely as possible to the conventional mathematics of differential geometry, in which spaces of tangent and cotangent vectors form (nontrivial) bundles over the space of primals. For example, the dimension of the space of (co)tangent vectors to a sum $\mathbb{R}^n \sqcup \mathbb{R}^m$ is either n or m , depending on whether the base point (primal) is chosen in the left or right component. An added advantage of this dependently typed approach is that it leads to a cleaner categorical story in which all η -laws are preserved by the AD transformations and standard categorical logical relations techniques can be used in the correctness proof.

That said, while the dependent types we presented give extra type safety that simplify mathematical foundations and the correctness argument underlying our AD techniques, nothing breaks if we keep the transformation on programs the same and simply coarse grain the types by removing any type dependency. This may be desirable in practical implementations of the

algorithms as most practical programming languages have either no or only limited support for type dependency.

To be precise, we can perform the following coarse-graining transformation $(-)^{\dagger}$ on the types of the target language, which removes all type dependency:

$$\begin{array}{llll}
 \underline{\alpha}^{\dagger} & \stackrel{\text{def}}{=} \underline{\alpha} & (\text{case } t \text{ of } \{\ell_1 x_1 \rightarrow \underline{\alpha}_1 \mid \dots \ell_n x_n \rightarrow \underline{\alpha}_n\})^{\dagger} & \stackrel{\text{def}}{=} \underline{\alpha}_1^{\dagger} \vee \dots \vee \underline{\alpha}_n^{\dagger} \\
 \mathbf{real}^{n\dagger} & \stackrel{\text{def}}{=} \mathbf{real}^n & (\underline{\mu}\underline{\alpha}.\underline{\alpha})^{\dagger} & \stackrel{\text{def}}{=} \underline{\mu}\underline{\alpha}.\underline{\alpha}^{\dagger} \\
 \mathbf{1}^{\dagger} & \stackrel{\text{def}}{=} \mathbf{1} & (\underline{\nu}\underline{\alpha}.\underline{\alpha})^{\dagger} & \stackrel{\text{def}}{=} \underline{\mu}\underline{\alpha}.\underline{\alpha}^{\dagger} \\
 (\underline{\alpha}*\underline{\sigma})^{\dagger} & \stackrel{\text{def}}{=} \underline{\alpha}^{\dagger}*\underline{\sigma}^{\dagger} & (\underline{\alpha} \multimap \underline{\sigma})^{\dagger} & \stackrel{\text{def}}{=} \underline{\alpha}^{\dagger} \multimap \underline{\sigma}^{\dagger} \\
 (\Pi x : \tau.\underline{\sigma})^{\dagger} & \stackrel{\text{def}}{=} \Pi x : \tau^{\dagger}.\underline{\sigma}^{\dagger} & (\Pi x : \tau.\sigma)^{\dagger} & \stackrel{\text{def}}{=} \Pi x : \tau^{\dagger}.\sigma^{\dagger} \\
 (\Sigma x : \tau.\underline{\sigma})^{\dagger} & \stackrel{\text{def}}{=} \Sigma x : \tau^{\dagger}.\underline{\sigma}^{\dagger} & (\Sigma x : \tau.\sigma)^{\dagger} & \stackrel{\text{def}}{=} \Sigma x : \tau^{\dagger}.\sigma^{\dagger}.
 \end{array}$$

In fact, seeing that $(\text{case } \ell_1 x_1 \rightarrow \underline{\alpha}_1 \mid \dots \ell_n x_n \rightarrow \underline{\alpha}_n \text{ of } t\{\})$ -types were the only source of type dependency in our language while these are translated to nondependent types, all Π - and Σ -types are simply translated to powers, copowers, function types and product types:

$$\begin{array}{ll}
 (\Pi x : \tau.\underline{\sigma})^{\dagger} = \tau^{\dagger} \rightarrow \underline{\sigma}^{\dagger} & (\Pi x : \tau.\sigma)^{\dagger} = \tau^{\dagger} \rightarrow \sigma^{\dagger} \\
 (\Sigma x : \tau.\underline{\sigma})^{\dagger} = !\tau^{\dagger} \otimes \underline{\sigma}^{\dagger} & (\Sigma x : \tau.\sigma)^{\dagger} = \tau^{\dagger} * \sigma^{\dagger}.
 \end{array}$$

Our translation $(-)^{\dagger}$ is the identity on programs.

The types $\underline{\alpha}_1 \vee \dots \vee \underline{\alpha}_n$ require some elaboration. We give this in the next section where we explain how to implement all required linear types and their terms in a standard functional programming language.

15.3 Removing linear types from the target language

15.3.1 Basics

As discussed in detail in Vákár and Smeding (2022), Vákár (2021) and demonstrated in the Haskell implementation available at <https://github.com/VMatthijs/CHAD>, the types \mathbf{real}^n , $\mathbf{1}$, $\underline{\alpha}*\underline{\sigma}$, $\tau \rightarrow \underline{\sigma}$, $!\tau \otimes \underline{\sigma}$, and $\underline{\alpha} \multimap \underline{\sigma}$ (and, obviously, the ordinary Cartesian function and product types $\tau \rightarrow \sigma$ and $\tau*\sigma$) together with their terms can all be implemented in a standard functional language. The core idea is to implement $\underline{\alpha}$ as the type $\underline{\alpha}^{\ddagger}$:

$$\begin{array}{ll}
 \mathbf{real}^{n\ddagger} & \stackrel{\text{def}}{=} \mathbf{real}^n & (\tau \rightarrow \underline{\sigma})^{\ddagger} & \stackrel{\text{def}}{=} \tau^{\ddagger} \rightarrow \underline{\sigma}^{\ddagger} \\
 \mathbf{1}^{\ddagger} & \stackrel{\text{def}}{=} \mathbf{1} & (!\tau \otimes \underline{\sigma})^{\ddagger} & \stackrel{\text{def}}{=} [(\tau^{\ddagger}, \underline{\sigma}^{\ddagger})] \\
 (\underline{\alpha}*\underline{\sigma})^{\ddagger} & \stackrel{\text{def}}{=} \underline{\alpha}^{\ddagger}*\underline{\sigma}^{\ddagger} & (\underline{\alpha} \multimap \underline{\sigma})^{\ddagger} & \stackrel{\text{def}}{=} \underline{\alpha}^{\ddagger} \rightarrow \underline{\sigma}^{\ddagger}.
 \end{array}$$

Crucially, we implement the copowers as abstract types that can under the hood be lists of pairs $[(\tau^{\ddagger}, \underline{\sigma}^{\ddagger})]$ and we implement the linear function types as abstract types that can under the hood be plain functions $\underline{\alpha}^{\ddagger} \rightarrow \underline{\sigma}^{\ddagger}$. As discussed in Vákár and Smeding (2022), Vákár (2021) and shown in the Haskell implementation, this translation extends to programs and leads to a correct implementation of CHAD on a simply typed λ -calculus.

We explain here how to extend this translation to implement the extra linear types $\underline{\alpha}_1 \vee \dots \vee \underline{\alpha}_n$, $\underline{\mu}\underline{\alpha}.\underline{\alpha}$ and $\underline{\nu}\underline{\alpha}.\underline{\alpha}$ required to perform AD on source languages that additionally use sum types, inductive types, and coinductive types.

15.3.2 Linear sum types $\underline{\alpha}_1 \vee \dots \vee \underline{\alpha}_n$

We briefly outline three possible implementations $(\underline{\alpha}_1 \vee \dots \vee \underline{\alpha}_n)^\ddagger$ of the linear sum types $\underline{\alpha}_1 \vee \dots \vee \underline{\alpha}_n$:

- (1) as a finite (bi)product $\underline{\alpha}_1^\ddagger * \dots * \underline{\alpha}_n^\ddagger$;
- (2) as a finite lifted sum $\{Zero \mid Opt_1 \underline{\alpha}_1^\ddagger \mid \dots \mid Opt_n \underline{\alpha}_n^\ddagger\}$;
- (3) as a finite sum $\{Opt_1 \underline{\alpha}_1^\ddagger \mid \dots \mid Opt_n \underline{\alpha}_n^\ddagger\}$.

Approach 1 has the advantage that we can keep the implementation total. As demonstrated in Appendix C, this allows us to easily extend the logical relations argument for the correctness of the applied implementation of Vákár and Smeding (2022) and Vákár (2021) (in actual Haskell, available at <https://github.com/VMatthijs/CHAD>). Categorically, what is going on is that, for a locally indexed category $\mathcal{L} : \mathcal{C}^{op} \rightarrow \mathbf{Cat}$ with indexed finite biproducts and \dashv -types, $(X_1 \sqcup \dots \sqcup X_n, A_1 \times \dots \times A_n)$ is a weak coproduct of $(X_1, A_1), \dots, (X_n, A_n)$ in both $\Sigma_{\mathcal{C}}\mathcal{L}$ and $\Sigma_{\mathcal{C}}\mathcal{L}^{op}$: that is, a coproduct for which the η -law may fail. The logical relations proof of Appendix C lifts these weak coproducts to the subscone, demonstrating that this implementation of CHAD for coproducts indeed computes semantically correct derivatives.

This approach was first implemented in the Haskell implementation of CHAD. However, a major downside of approach 1 is its inefficiency: it represents (co)tangents to a coproducts as tuples of (co)tangents to the component spaces, all but one of which are known to be zero. This motivates approaches 2 and 3.

Approach 2 exploits this knowledge that all but one component of the (co)tangent space are zero by only storing the single nonzero component, corresponding to the connected component the current primal is in. To see the correctness of this approach, we can add an extra error element \perp to all our linear types $\overrightarrow{\mathcal{D}}(\tau)_2$ and $\overleftarrow{\mathcal{D}}(\tau)_2$, for which $\perp + x = \perp$, and do a manual (total) logical relations proof. We can then note that we can also leave out the error element of the data type and throw actual errors at runtime.

We pay for this more efficient representation in two ways:

- addition on the (co)tangent space is defined by:

$$Zero + x = x \quad x + Zero = x \quad Opt_i(t) + Opt_i(s) = Opt_i(t + s)$$

and hence is a partial operation that throws an error if we try to add $Opt_i(t) + Opt_j(s)$ for $i \neq j$;

- we need to add a new zero element *Zero* rather than simply reusing the zeros $Opt_i(\underline{0})$ that are present in each of the components, which should be equivalent for all practical purposes.

The first issue is not a problem at all in practice, as the more precise dependent types we have erased guarantee that CHAD only ever adds (co)tangents in the same component, meaning that the error can never be triggered in practice. However, it requires us to do a manual logical relations proof of correctness. This is the approach that is currently implemented in the reference Haskell implementation of CHAD. The second issue is a minor inefficiency that can become more serious if (co)inductive types are built using this representation of coproducts. This motivates approach 3.

Approach 3 addresses the second issue with approach 2 by removing the unnecessary extra element *Zero* of the (co)tangent spaces. To achieve this, however, the zeros $\underline{0}$ at each type $\overrightarrow{\mathcal{D}}(\tau)_2$ of tangent and $\overleftarrow{\mathcal{D}}(\tau)_2$ of cotangents need to be made functions $\underline{0} : \overrightarrow{\mathcal{D}}(\tau)_1 \rightarrow \overrightarrow{\mathcal{D}}(\tau)_2$ and $\underline{0} : \overleftarrow{\mathcal{D}}(\tau)_1 \rightarrow \overleftarrow{\mathcal{D}}(\tau)_2$, rather than mere constant zeros. Whenever the a zero is used by CHAD, it is called on the corresponding primal value that specifies in which component we want the zero to

land. While a mathematical formalization of this approach remains future work, we have shown this approach to work well in practice in an experimental Haskell implementation of CHAD. As we plan to detail in an applied follow-up paper, this approach also gives an efficient way of applying CHAD to dynamically sized arrays.

15.3.3 Linear inductive and coinductive types $\underline{\mu\alpha}.\alpha$ and $\underline{\nu\alpha}.\alpha$

As we have seen, linear coinductive types arise in reverse CHAD of inductive types as well as in forward CHAD of coinductive types. Similarly, linear inductive types arise in reverse CHAD of coinductive types as well as in forward CHAD of inductive types. It remains to be investigated how these can be best implemented. However, as was the case for the implementation of copowers and linear sum types, we are hopeful that the concrete denotational semantics can guide us

Observe that all polynomials $F : \mathbf{Vect} \rightarrow \mathbf{Vect}$ are of the form $W \mapsto L(A) + W^n$, where $L \dashv U : \mathbf{Set} \rightarrow \mathbf{Vect}$ is the usual free-forgetful adjunction. Therefore, $U \circ F = H \circ U$ for the polynomial $H : \mathbf{Set} \rightarrow \mathbf{Set}$ defined by $S \mapsto U(L(A)) \times S^n$. As the forgetful functor $F : \mathbf{Vect} \rightarrow \mathbf{Set}$ is monadic, it creates terminal coalgebras, hence $U(\nu F) = \nu H$. This suggests that we might be able to implement $(\underline{\nu\alpha}.\alpha)^\ddagger$ as the plain coinductive type $\underline{\nu\alpha}.\alpha^\ddagger$, where $\alpha^\ddagger \stackrel{\text{def}}{=} \alpha$.

Similarly, we have that $F \circ L = L \circ E$ for the polynomial $E : \mathbf{Set} \rightarrow \mathbf{Set}$ defined by $E(X) = A \sqcup \bigsqcup_n X$. Therefore, we have that $\mu F = L(\mu E) = (\mu E) \rightarrow \mathbb{R}$. This suggests that the implementation of linear inductive types might be achieved by “delinearizing” a polynomial F to E , taking the initial algebra of E and taking the function type to \mathbb{R} .

We are hopeful that this theory will lead to a practical implementation, but the details remain to be verified.

16. Related Work

Automatic differentiation has long been studied by the scientific computing community. In fact, its study goes back many decades with forward-mode AD being introduced by Wengert (1964) and variants of reverse-mode AD seemingly being reinvented several times, for example, by Linnainmaa (1970) and Speelpenning (1980). For brief reviews of this complex history and the basic ideas behind AD, we refer the reader to Baydin et al. (2017). For a more comprehensive account of the traditional work on AD, see the standard reference text (Griewank and Walther 2008).

In this section, we focus, instead, on the more recent work that has proliferated since the programming languages community started seriously studying AD. Their objectives are more closely aligned with those of the present paper.

Pearlmutter and Siskind (2008) is one of the early programming languages papers trying to extend the scope of AD from the traditional setting of first-order imperative languages to more expressive programming languages. Specifically, this applied paper proposes a method to use reverse-mode AD on an untyped higher-order functional language, through the use of an intricate source code transformation that employs ideas similar to defunctionalization. It focuses on implementation rather than correctness or intended semantics. Alvarez-Picallo et al. (2023) recently simplified this code transformation and formalized its correctness.

Prompted by Plotkin (2018), there has, more recently, been a push in the programming language community to learn from Pearlmutter and Siskind (2008) and arrive at a definition of (reverse) AD as a source code transformation on expressive languages that should ideally be simple, semantically motivated and correct, compositional and efficient.

Among this work, Wang et al. (2019) specifies and implements much simpler reverse AD transformation on a higher-order functional language with sum types. The price they have to pay is that the transformation relies on the use of delimited continuations in the target language.

Various more theoretical works give formalizations and correctness proofs of reverse AD on expressive languages through the use of custom operational semantics. Abadi and Plotkin (2020) gives such an analysis for a first-order functional language with recursion, using an operational semantics that mirrors the runtime tracing techniques used in practice. Mak and Ong (2020) instead works with a total higher-order language that is a variant of the differential λ -calculus. Using slightly different operational techniques, coming from linear logic, Brunel et al. (2020) and Mazza and Pagani (2021) give an analysis of reverse AD on a simply typed λ -calculus and programmable computable functions. Notably, Brunel et al. (2020) shows that their algorithm has the right complexity if one assumes a specific operational semantics for their linear λ -calculus with what they call a “linear factoring rule.” Very recently, Krawiec et al. (2022) applied the idea of reverse AD through tracing to a higher-order functional language with variant types. They implement the custom operational semantics as an evaluator and give a denotational correctness proof (using logical relations techniques similar to those of Barthe et al. 2020; Huot et al. 2020) as well as an asymptotic complexity proof about the full code transformation plus evaluator.

Elliott (2018) takes a different approach that is much closer to the present paper by working with a target language that is a plain functional language and does not depend on a custom operational semantics or an evaluator for traces. Although this approach also naturally has linear types, it is a fundamentally different algorithm from that of Brunel et al. (2020) and Mazza and Pagani (2021): for example, the linear types can be coarse-grained to plain simply typed code (e.g., Haskell) with the right computational complexity, even under the standard operational semantics of functional languages. This is the approach that we have been referring to as CHAD. Elliott’s CHAD transformation, however, is restricted to a first-order functional language with tuples. Vytiniotis et al. (2019) and Vákár (2021) both present (the same) extensions of CHAD to apply to a higher-order functional source language, while still working with a functional target language. While Vytiniotis et al. (2019) relates CHAD to the approach of Alvarez-Picallo et al. (2023) and Pearlmuter and Siskind (2008), Vákár (2021) and its extended version (Vákár and Smeding 2022) give a (denotational) semantic foundation and correctness proof for CHAD, using a combination of logical relations techniques that Barthe et al. (2020) and Huot et al. (2022, 2020) had previously used to prove correct (higher-order) forward-mode AD together with the observation that AD can be understood through the framework of lenses or Grothendieck fibrations, which had previously been made by Fong et al. (2019) and Cockett et al. (2020). The present paper extends CHAD to further apply to source languages with variant types and (co)inductive types. To our knowledge, it is the first paper to consider reverse AD on languages with such expressive type systems.

Acknowledgements. This project has received funding from the European Union’s Horizon 2020 research and innovation program under the Marie Skłodowska-Curie grant agreement No. 895827 and from the Nederlandse Organisatie voor Wetenschappelijk Onderzoek under NWO Veni grant number VI.Veni.202.124. This research was also supported through the program “Oberwolfach Leibniz Fellows” by the Mathematisches Forschungsinstitut Oberwolfach in 2022 and partially supported by the CMUC, Centre for Mathematics of the University of Coimbra – UIDB/00324/2020, funded by the Portuguese Government through FCT/MCTES.

We thank Tom Smeding, Gordon Plotkin, Wouter Swierstra, Gabriele Keller, Ohad Kammar, Dimitrios Vytiniotis, Patricia Johann, Michelle Pagani, Michael Betancourt, Bob Carpenter, Sam Staton, Mathieu Huot, Curtis Chin Jen Sem and Amir Shaikhha for helpful discussions about topics related to the present work.

Notes

I In fact, the (co)tangent vectors form a vector space and (transposed) derivatives are vector space homomorphisms. Surprisingly, it is only the monoid structure that is relevant to phrasing and proving correct CHAD. Therefore, we choose to emphasize this monoid structure over the full vector space structure. For example, CHAD-like algorithms also works for more general data types than the real numbers, as long as they form a commutative monoid. An interesting example is a datatype that implements saturation arithmetic, as is commonly used as a cheap alternative to floating point arithmetic in machine learning.

- 2 In the case of tangent vectors, this often presented in terms of the (equivalent) induced lift $(\Sigma_{x \in X} \mathcal{T}_x X) \rightarrow (\Sigma_{y \in Y} \mathcal{T}_y Y)$ of $f : X \rightarrow Y$ to the tangent bundles.
- 3 In fact, on such infinite-dimensional spaces, we have many inequivalent definitions of derivative (that all coincide for finite-dimensional spaces) (Christensen and Wu 2014; Iglesias-Zemmour 2013).
- 4 This is a generalization of the proof given in Vákár (2021), where the result is established for locally indexed categories.
- 5 We could have allowed nonstrict preservation but, in our context, it is more practical to keep things as strict as possible.
- 6 Nothing would stop us from defining the derivative of a primitive operations as a more general term, rather than a linear operation. In fact, that is what we considered in Vákár and Smeding (2022), Vákár (2021). However, we believe that treating derivatives of operations as linear operations slightly simplifies the development and is no limitation, seeing that we are free to implement linear operations as we please in a practical AD system.
- 7 The basic definition of Kan extension can be found, for instance, in Mac Lane (1971, Chapter X). Although one can verify it directly, (64) follows from the general result about pointwise Kan extensions; see, for instance, Dubuc (1970) or Kelly (2005, Chapter 4).
- 8 Some of the results presented here hold under slightly more general conditions. But we chose to make the most of our setting, which is general enough for our proof and many others cases of interest.
- 9 The original result on adjoint triangles was proven in Dubuc (1968). Further comments and generalizations are given in Lucatelli Nunes (2018), while a precise statement for our case is given in Lucatelli Nunes (2016, Corollary 1.2).
- 10 For the original statement, please refer to Kelly (1974). For the general case of lax algebras, see, for instance, Lucatelli Nunes (2017, Corollary 1.4.15).
- 11 We even claim that the result is useful when the semantics of the primitive operations is not differentiable everywhere in the domain; see the revised version of Lucatelli Nunes and Vákár (2022a).

References

- Abadi, M. and Plotkin, G. D. (2020). A simple differentiable programming language. In: *Proceedings of POPL 2020*, ACM.
- Adámek, J. and Koubek, V. (1979). Least fixed point of a functor. *Journal of Computer and System Sciences* **19** (2) 163–178.
- Adámek, J., Milius, S. and Moss, L. (2010). *Initial Algebras and Terminal Coalgebras: A Survey*. https://web.archive.org/web/20150919161434/https://www.tu-braunschweig.de/Medien-DB/iti/survey_full.pdf.
- Adámek, J. and Rosický, J. (1994). *Locally Presentable and Accessible Categories*, London Mathematical Society Lecture Note Series, vol. 189, Cambridge, Cambridge University Press.
- Adámek, J. and Rosický, J. (2020). How nice are free completions of categories? *Topology and Its Applications* **273** 24. Id/No 106972.
- Ahman, D., Ghani, N. and Plotkin, G. D. (2016). Dependent types and fibred computational effects. In: *International Conference on Foundations of Software Science and Computation Structures*, Springer, 36–54.
- Altenkirch, T., Levy, P. and Staton, S. (2010). Higher-order containers. In: *Conference on Computability in Europe*, Springer, 11–20.
- Alvarez-Picallo, M., Ghica, D. R., Sprunger, D. and Zanasi, F. (2023). Functorial string diagrams for reverse-mode automatic differentiation. In: Klin, B. and Pimentel, E. (eds.) *31st EACSL Annual Conference on Computer Science Logic, CSL 2023, February 13–16, 2023, Warsaw, Poland*, LIPIcs, vol. 252, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 6:1–6:20.
- Barr, M. (1993). Terminal coalgebras in well-founded set theory. *Theoretical Computer Science* **114** (2) 299–315.
- Barr, M. and Wells, C. (2005). Toposes, triples and theories. *Representation Theory Application Categories* **2005** (12) 1–288.
- Barthe, G., Crubillé, R., Lago, U. D. and Gavazzo, F. (2020). On the versatility of open logical relations - continuity, automatic differentiation, and a containment theorem. In: Müller, P. (ed.) *Programming Languages and Systems - 29th European Symposium on Programming, ESOP 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25–30, 2020, Proceedings*, Lecture Notes in Computer Science, vol. 12075, Springer, 56–83.
- Baydin, A. G., Pearlmutter, B. A., Radul, A. A. and Siskind, J. M. (2017). Automatic differentiation in machine learning: a survey. *Journal of Machine Learning Research* **18** 153:1–153:43.
- Bird, G. (1984). Limits in 2-categories of locally-presentable categories. *Sydney Category Seminar Report*. Phd thesis, University of Sydney.
- Borceux, F. and Janelidze, G. (2001). *Galois Theories*, Cambridge Studies in Advanced Mathematics, vol. 72, Cambridge, Cambridge University Press.
- Brunel, A., Mazza, D. and Pagani, M. (2020). Backpropagation in the simply typed lambda-calculus with linear negation. In: *Proceedings of POPL 2020*.
- Carboni, A., Lack, S. and Walters, R. F. C. (1993). Introduction to extensive and distributive categories. *Journal of Pure and Applied Algebra* **84** (2) 145–158.
- Christensen, J. D. and Wu, E. (2014). Tangent spaces and tangent bundles for diffeological spaces. arXiv preprint arXiv:1411.5425.
- Cockett, J. R. B., Cruttwell, G. S. H., Gallagher, J., Lemay, J.-S. P., MacAdam, B., Plotkin, G. D. and Pronk, D. (2020). Reverse derivative categories. In: *Proceedings of CSL 2020*.

- Crole, R. L. (1993). *Categories for Types*, Cambridge, Cambridge University Press.
- Diller, J. (1974). Eine variante zur dialectica-interpretation der heyting-arithmetik endlicher typen. *Archiv für mathematische Logik und Grundlagenforschung* **16** (1–2) 49–66.
- Dubuc, E. (1968). Adjoint triangles. In: *Reports of the Midwest Category Seminar, II*, Berlin, Springer, 69–91.
- Dubuc, E. (1970). *Kan Extensions in Enriched Category Theory*, Lecture Notes in Mathematics, vol. 145, Cham, Springer.
- Elliott, C. (2018). The simple essence of automatic differentiation. *Proceedings of the ACM on Programming Languages* **2** (ICFP) 70.
- Fong, B., Spivak, D. and Tuyéras, R. (2019). Backprop as functor: a compositional perspective on supervised learning. In: *2019 34th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, IEEE, 1–13.
- Gabriel, P. and Ulmer, F. (1971). *Lokal präsentierbare Kategorien. (Locally Presentable Categories)*, Lecture Notes in Mathematics, vol. 221, Cham, Springer.
- Gödel, V. K. (1958). Über eine bisher noch nicht benützte erweiterung des finiten standpunktes. *Dialectica* **12** (3–4) 280–287.
- Gray, J. W. (1966). Fibréd and cofibréd categories. In: *Proceedings of the Conference on Categorical Algebra (La Jolla, California, 1965)*, New York, Springer, 21–83.
- Griewank, A. and Walther, A. (2008). *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*, vol. 105, SIAM.
- Huot, M., Staton, S. and Vákár, M. (2022). Higher order automatic differentiation of higher order functions. *Logical Methods in Computer Science* **18** (1) 1–41.
- Huot, M., Staton, S. and Vákár, M. (2020). Correctness of automatic differentiation via diffeologies and categorical gluing. In: *Proceedings of FoSSaCS*.
- Hyland, J. M. E. (2002). Proof theory in the abstract. *Annals of Pure and Applied Logic* **114** (1–3) 43–78.
- Iglesias-Zemmour, P. (2013). *Diffeology*, American Mathematical Society.
- Jacobs, B. (1999). *Categorical Logic and Type Theory*, Studies in Logic and the Foundations of Mathematics, vol. 141 Amsterdam, Elsevier.
- Johnstone, P. T. (2002). *Sketches of an Elephant: A Topos Theory Compendium*, vol. 2, Oxford, Oxford University Press.
- Kelly, G. M. (1974). Doctrinal adjunction. In: *Category Seminar (Proceedings Sydney Category Theory Seminar, 1972/1973)*, Lecture Notes in Mathematics, vol. 420, 257–280.
- Kelly, G. M. (2005). Basic concepts of enriched category theory. *Representation Theory Application Categories* **2005** (10) 1–136.
- Kerjean, M. and Pédrot, P.-M. (2021). ∂ is for Dialectica: Typing Differentiable Programming. Working Paper or Preprint.
- Krawiec, F., Peyton Jones, S., Krishnaswami, N., Ellis, T., Eisenberg, R. A. and Fitzgibbon, A. (2022). Provably correct, asymptotically efficient, higher-order reverse-mode automatic differentiation. *Proceedings of the ACM on Programming Languages* **6** (POPL) 1–30.
- Lack, S. (2012). Non-canonical isomorphisms. *Journal of Pure and Applied Algebra* **216** (3) 593–597.
- Lambek, J. and Scott, P. J. (1988). *Introduction to Higher-Order Categorical Logic*, vol. 7, Cambridge, Cambridge University Press.
- Lee, J. M. (2013). Smooth manifolds. In: *Introduction to Smooth Manifolds*, Springer, 1–31.
- Leinster, T. (2014). *Basic Category Theory*, vol. 143, Cambridge Studies in Advanced Mathematics, Cambridge, Cambridge University Press.
- Linnainmaa, S. (1970). *The Representation of the Cumulative Rounding Error of an Algorithm as a Taylor Expansion of the Local Rounding Errors. Master's thesis (in Finnish), Univ. Helsinki*, 6–7.
- Lucatelli Nunes, F. (2016). On biadjoint triangles. *Theory and Applications of Categories* **31** Paper No. 9 217–256.
- Lucatelli Nunes, F. (2017). *Pseudomonads and Descent*. Phd thesis (Chapter 1). University of Coimbra. arXiv: [1802.01767](https://arxiv.org/abs/1802.01767).
- Lucatelli Nunes, F. (2018). On lifting of biadjoints and lax algebras. *Categories and General Algebraic Structures with Applications* **9** (1) 29–58.
- Lucatelli Nunes, F. (2019). Pseudoalgebras and non-canonical isomorphisms. *Applied Categorical Structures* **27** (1) 55–63.
- Lucatelli Nunes, F. (2021). Descent data and absolute Kan extensions. *Theory and Applications of Categories* **37** Paper No. 18 530–561.
- Lucatelli Nunes, F. (2022). Semantic factorization and descent. *Applied Categorical Structures* **30** (6) 1393–1433.
- Lucatelli Nunes, F. and Vákár, M. (2022a). Automatic Differentiation for ML-family languages: correctness via logical relations. arXiv e-prints, arXiv:[2210.07724](https://arxiv.org/abs/2210.07724).
- Lucatelli Nunes, F. and Vákár, M. (2022b). Logical Relations for Partial Features and Automatic Differentiation Correctness. arXiv e-prints, arXiv:[2210.08530](https://arxiv.org/abs/2210.08530).
- Mac Lane, S. (1971). *Categories for the Working Mathematician*, Graduate Texts in Mathematics, vol. 5, Cham, Springer.
- MacDonald, J. and Sobral, M. (2004). Aspects of monads. In: *Categorical Foundations*, Encyclopedia of Mathematics and its Applications, vol. 97, Cambridge, Cambridge University Press, 213–268.
- Mak, C. and Ong, L. (2020). A differential-form pullback programming language for higher-order reverse-mode automatic differentiation. arxiv:[2002.08241](https://arxiv.org/abs/2002.08241).
- Makkai, M. and Paré, R. (1989). *Accessible Categories: The Foundations of Categorical Model Theory*, Contemporary Mathematics, vol. 104 Providence, RI, American Mathematical Society.

- Mazza, D. and Pagani, M. (2021). Automatic differentiation in pcf. *Proceedings of the ACM on Programming Languages* 5 (POPL) 1–27.
- Moss, S. K. and von Glehn, T. (2018). Dialectica models of type theory. In: Dawar, A. and Grädel, E. (eds.) *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09–12, 2018*, ACM, 739–748.
- Pearlmutter, B. A. and Siskind, J. M. (2008). Reverse-mode AD in a functional framework: Lambda the ultimate backpropagator. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 30 (2) 7.
- Pitts, A. M. (1995). Categorical logic. Technical report, University of Cambridge, Computer Laboratory.
- Plotkin, G. (2018). Some principles of differential programming languages. *Invited talk, POPL*.
- Santocanale, L. (2002). μ -bicomplete categories and parity games. *RAIRO - Theoretical Informatics and Applications - Informatique Théorique et Applications* 36 (2) 195–227.
- Speelpenning, B. (1980). Compiling fast partial derivatives of functions given by algorithms. Technical report, Illinois University, Urbana (USA). Department of Computer Science.
- Tu, L. W. (2011). Manifolds. In: *An Introduction to Manifolds*, Springer, 47–83.
- Vákár, M. (2017). *In Search of Effectful Dependent Types*. Phd thesis, University of Oxford. arXiv preprint arXiv:1706.07997.
- Vákár, M. (2021). Reverse AD at higher types: pure, principled and denotationally correct. In: *ESOP*, 607–634.
- Vákár, M. and Smeding, T. (2022). CHAD: combinatory homomorphic automatic differentiation. *ACM Transactions on Programming Languages and Systems* 44 (3) 20:1–20:49.
- Vytiniotis, D., Belov, D., Wei, R., Plotkin, G. and Abadi, M. (2019). The differentiable curry. Program Transformations for ML Workshop at NeurIPS 2019. <https://openreview.net/forum?id=ryxuz9SzDB>.
- Wang, F., Wu, X., Essertel, G., Decker, J. and Rompf, T. (2019). Demystifying differentiable programming: shift/reset the penultimate backpropagator. *Proceedings of the ACM on Programming Languages* 3 (ICFP) 1–31.
- Wengert, R. E. (1964). A simple automatic derivative evaluation program. *Communications of the ACM* 7 (8) 463–464.

Appendix A. Pseudo-Preterminal Objects in **Cat**

The appropriate two-dimensional analogue to preterminal objects are the pseudo-preterminal ones. Namely, in the case of **Cat**:

Definition 131. *An object W in **Cat** is pseudo-preterminal if the category of functors $\mathbf{Cat}[X, W]$ is a groupoid for any object X in **Cat**.*

Lemma 132 establishes that the initial and terminal categories are, up to equivalence, the only pseudo-preterminal objects of **Cat**.

Lemma 132 (Pseudo-preterminal objects in **Cat**). *Let W be an object of **Cat**. Assuming that W is not the initial object of **Cat**, the following statements are equivalent:*

- i The unique functor $W \rightarrow \mathbb{1}$ is an equivalence.*
- ii The projection $\pi_W : W \times W \rightarrow W$ is an equivalence.*
- iii The identity $\text{id}_W : W \rightarrow W$ is naturally isomorphic to a constant functor $c : W \rightarrow W$.*
- iv If $f, g : X \rightarrow W$ are functors, then there is a natural isomorphism $f \cong g$ (that is to say, W is pseudo-preterminal).*

Proof. Assuming (i), denoting by $t : W \rightarrow \mathbb{1}$ the unique functor, we have that π_W is the composition $W \times W \xrightarrow{\text{id}_W \times t} W \times \mathbb{1} \cong W$. Hence, since id_W and t are equivalences, we conclude that π_W is an equivalence. This proves that (i) \Rightarrow (ii).

Given any constant functor $c : W \rightarrow W$, we have that $(\text{id}_W, c) : W \rightarrow W \times W$ and the diagonal functor $(\text{id}_W, \text{id}_W) : W \rightarrow W \times W$ are such that $\pi_W \circ (\text{id}_W, c) = \text{id}_W$ and $\pi_W \circ (\text{id}_W, \text{id}_W) = \text{id}_W$. Hence, assuming (ii), we have that (id_W, c) and $(\text{id}_W, \text{id}_W)$ are inverse equivalences of π_W . Thus we have a natural isomorphism $(\text{id}_W, c) \cong (\text{id}_W, \text{id}_W)$ which implies that

$$c \cong \pi_2 \circ (\text{id}_W, c) \cong \pi_2 \circ (\text{id}_W, \text{id}_W) \cong \text{id}_W.$$

This proves that (ii) \Rightarrow (iii).

Assuming (iii), if $f, g : X \rightarrow W$ are functors, we have the natural isomorphisms:

$$f = \text{id}_W \circ f \cong c \circ f = c \circ g \cong \text{id}_W \circ g = g.$$

This shows that (iii) \Rightarrow (iv).

Finally, assuming (iv), we have that, given any functor $c : \mathbb{1} \rightarrow W$, the composition $W \rightarrow \mathbb{1} \xrightarrow{c} W$ is naturally isomorphic to the identity. Hence, $W \rightarrow \mathbb{1}$ is an equivalence. This shows that (iv) \Rightarrow (i). \square

Remark 133. The equivalence (ii) \Leftrightarrow (iv) holds for the general context of any 2-category. The other equivalences mean that $\mathbb{1}$ and $\mathbb{0}$ are, up to equivalence, the unique pseudo-preterminal objects of **Cat**. The reader might compare the result, for instance, with the characterization of contractible spaces in basic homotopy theory.

Appendix B. CHAD Transformation without Sharing Between Primal and (Co)tangents

In this section, we list the CHAD program transformations $\vec{\mathcal{D}}(\Gamma)_1 \vdash \vec{\mathcal{D}}_{\Gamma}(t)_1 : \vec{\mathcal{D}}(\tau)$, $\vec{\mathcal{D}}(\Gamma)_1; v : \vec{\mathcal{D}}(\Gamma)_2 \vdash \vec{\mathcal{D}}_{\Gamma}(t)_2 : \vec{\mathcal{D}}(\tau)_2[\vec{\mathcal{D}}_{\Gamma}(t)_1/\rho]$, $\overleftarrow{\mathcal{D}}(\Gamma)_1 \vdash \overleftarrow{\mathcal{D}}_{\Gamma}(t)_1 : \overleftarrow{\mathcal{D}}(\tau)$ and $\overleftarrow{\mathcal{D}}(\Gamma)_1; v : \overleftarrow{\mathcal{D}}(\tau)_2[\overleftarrow{\mathcal{D}}_{\Gamma}(t)_1/\rho] \vdash \overleftarrow{\mathcal{D}}_{\Gamma}(t)_2 : \overleftarrow{\mathcal{D}}(\Gamma)_2$ of a program $\Gamma \vdash t : \tau$ that keep the primals and (co)tangents separate without sharing computation. We advise against implementing these, due to

- (1) the code explosion they can result in, leading to a potentially large code size and compilation times;
- (2) the lack of sharing of computation they can result in, leading to poor runtime performance.

B.1 Forward-mode AD

$$\vec{\mathcal{D}}_{\Gamma}(\text{op}(t_1, \dots, t_k))_1 \stackrel{\text{def}}{=} \mathbf{let } x_1 = \vec{\mathcal{D}}_{\Gamma}(t_1)_1 \mathbf{ in } \dots \mathbf{ let } x_k = \vec{\mathcal{D}}_{\Gamma}(t_k)_1 \mathbf{ in op}(x_1, \dots, x_k)$$

$$\vec{\mathcal{D}}_{\Gamma}(x)_1 \stackrel{\text{def}}{=} x$$

$$\vec{\mathcal{D}}_{\Gamma}(\mathbf{let } x = t \mathbf{ in } s)_1 \stackrel{\text{def}}{=} \mathbf{let } x = \vec{\mathcal{D}}_{\Gamma}(t)_1 \mathbf{ in } \vec{\mathcal{D}}_{\Gamma, x}(s)_1$$

$$\vec{\mathcal{D}}_{\Gamma}(\langle \rangle)_1 \stackrel{\text{def}}{=} \langle \rangle$$

$$\vec{\mathcal{D}}_{\Gamma}(\langle t, s \rangle)_1 \stackrel{\text{def}}{=} \langle \vec{\mathcal{D}}_{\Gamma}(t)_1, \vec{\mathcal{D}}_{\Gamma}(s)_1 \rangle$$

$$\vec{\mathcal{D}}_{\Gamma}(\mathbf{fst } (t))_1 \stackrel{\text{def}}{=} \mathbf{fst } (\vec{\mathcal{D}}_{\Gamma}(t)_1)$$

$$\vec{\mathcal{D}}_{\Gamma}(\mathbf{snd } (t))_1 \stackrel{\text{def}}{=} \mathbf{snd } (\vec{\mathcal{D}}_{\Gamma}(t)_1)$$

$$\vec{\mathcal{D}}_{\Gamma}(\lambda x. t)_1 \stackrel{\text{def}}{=} \lambda x. \langle \vec{\mathcal{D}}_{\Gamma, x}(t)_1, \lambda v. \mathbf{let } v = \langle \mathbb{0}, v \rangle \mathbf{ in } \vec{\mathcal{D}}_{\Gamma, x}(t)_2 \rangle$$

$$\vec{\mathcal{D}}_{\Gamma}(t s)_1 \stackrel{\text{def}}{=} \mathbf{fst } (\vec{\mathcal{D}}_{\Gamma}(t)_1 \vec{\mathcal{D}}_{\Gamma}(s)_1)$$

$$\vec{\mathcal{D}}_{\Gamma}(\ell t)_1 \stackrel{\text{def}}{=} \ell(\vec{\mathcal{D}}_{\Gamma}(t)_1)$$

$$\vec{\mathcal{D}}_{\Gamma}(\mathbf{case } t \mathbf{ of } \{ \ell_1 x_1 \rightarrow s_1 \mid \dots \mid \ell_n x_n \rightarrow s_n \})_1 \stackrel{\text{def}}{=}$$

$$\mathbf{case } \vec{\mathcal{D}}_{\Gamma_1} \mathbf{ of } \{ \ell_1 x_1 \rightarrow \vec{\mathcal{D}}_{\Gamma_1, x_1}(s_1)_1 \mid \dots \mid \ell_n x_n \rightarrow \vec{\mathcal{D}}_{\Gamma_1, x_n}(s_n)_1 \} (t)$$

$$\begin{aligned}
 \vec{\mathcal{D}}_{\bar{\Gamma}}(\mathbf{roll} \ t)_1 &\stackrel{\text{def}}{=} \mathbf{roll} \ \vec{\mathcal{D}}_{\bar{\Gamma}}(t)_1 \\
 \vec{\mathcal{D}}_{\bar{\Gamma}}(\mathbf{fold} \ t \ \mathbf{with} \ x \rightarrow s)_1 &\stackrel{\text{def}}{=} \mathbf{fold} \ \vec{\mathcal{D}}_{\bar{\Gamma}}(t)_1 \ \mathbf{with} \ x \rightarrow \vec{\mathcal{D}}_x]s_1 \\
 \vec{\mathcal{D}}_{\bar{\Gamma}}(\mathbf{gen} \ \mathbf{from} \ t \ \mathbf{with} \ x \rightarrow s)_1 &\stackrel{\text{def}}{=} \mathbf{gen} \ \mathbf{from} \ \vec{\mathcal{D}}_{\bar{\Gamma}}(t)_1 \ \mathbf{with} \ x \rightarrow \vec{\mathcal{D}}_x]s_1 \\
 \vec{\mathcal{D}}_{\bar{\Gamma}}(\mathbf{unroll} \ t)_1 &\stackrel{\text{def}}{=} \mathbf{unroll} \ \vec{\mathcal{D}}_{\bar{\Gamma}}(t)_1 \\
 \\
 \vec{\mathcal{D}}_{\bar{\Gamma}}(\mathbf{op}(t_1, \dots, t_k))_2 &\stackrel{\text{def}}{=} \mathbf{let} \ x_1 = \vec{\mathcal{D}}_{\bar{\Gamma}}(t_{11}) \ \mathbf{in} \ \dots \ \mathbf{let} \ x_k = \vec{\mathcal{D}}_{\bar{\Gamma}}(t_k)_1 \ \mathbf{in} \\
 &\quad \text{Dop}(x_1, \dots, x_k; \langle \vec{\mathcal{D}}_{\bar{\Gamma}}(t_1)_2 \bullet v, \dots, \vec{\mathcal{D}}_{\bar{\Gamma}}(t_k) \bullet_2 v \rangle) \\
 \vec{\mathcal{D}}_{\bar{\Gamma}}(x)_2 &\stackrel{\text{def}}{=} \mathbf{proj}_{\text{idx}(x; \bar{\Gamma})} (v) \\
 \vec{\mathcal{D}}_{\bar{\Gamma}}(\mathbf{let} \ x = t \ \mathbf{in} \ s)_2 &\stackrel{\text{def}}{=} \mathbf{let} \ x = \vec{\mathcal{D}}_{\bar{\Gamma}}(t)_1 \ \mathbf{in} \ \mathbf{let} \ v = \langle v, \vec{\mathcal{D}}_{\bar{\Gamma}}(t)_2 \rangle \ \mathbf{in} \ \vec{\mathcal{D}}_{\bar{\Gamma}, x}(s)_2 \\
 \vec{\mathcal{D}}_{\bar{\Gamma}}(\langle \rangle)_2 &\stackrel{\text{def}}{=} \langle \rangle \\
 \vec{\mathcal{D}}_{\bar{\Gamma}}(\langle t, s \rangle)_2 &\stackrel{\text{def}}{=} \langle \vec{\mathcal{D}}_{\bar{\Gamma}}(t)_2, \vec{\mathcal{D}}_{\bar{\Gamma}}(s)_2 \rangle \\
 \vec{\mathcal{D}}_{\bar{\Gamma}}(\mathbf{fst} \ (t))_2 &\stackrel{\text{def}}{=} \mathbf{fst} \ (\vec{\mathcal{D}}_{\bar{\Gamma}}(t)_2) \\
 \vec{\mathcal{D}}_{\bar{\Gamma}}(\mathbf{snd} \ (t))_2 &\stackrel{\text{def}}{=} \mathbf{snd} \ (\vec{\mathcal{D}}_{\bar{\Gamma}}(t)_2) \\
 \vec{\mathcal{D}}_{\bar{\Gamma}}(\lambda x. t)_2 &\stackrel{\text{def}}{=} \lambda x. \mathbf{let} \ v = \langle v, \emptyset \rangle \ \mathbf{in} \ \vec{\mathcal{D}}_{\bar{\Gamma}, x}(t)_2 \\
 \vec{\mathcal{D}}_{\bar{\Gamma}}(t \ s)_2 &\stackrel{\text{def}}{=} \mathbf{let} \ y = \vec{\mathcal{D}}_{\bar{\Gamma}}(s)_1 \ \mathbf{in} \ \vec{\mathcal{D}}_{\bar{\Gamma}}(t)_2 \ y + (\mathbf{snd} \ (\vec{\mathcal{D}}_{\bar{\Gamma}}(t)_1 \ y)) \bullet \vec{\mathcal{D}}_{\bar{\Gamma}}(s)_2 \\
 \vec{\mathcal{D}}_{\bar{\Gamma}}(\ell t)_2 &\stackrel{\text{def}}{=} \vec{\mathcal{D}}_{\bar{\Gamma}}(t)_2 \\
 \vec{\mathcal{D}}_{\bar{\Gamma}}(\mathbf{case} \ t \ \mathbf{of} \ \{\ell_1 x_1 \rightarrow s_1 \mid \dots \mid \ell_n x_n \rightarrow s_n\})_2 &\stackrel{\text{def}}{=} \\
 &\quad \mathbf{let} \ v = \langle v, \vec{\mathcal{D}}_{\bar{\Gamma}}(t)_2 \rangle \ \mathbf{in} \ \mathbf{case} \ \vec{\mathcal{D}}_{\bar{\Gamma}}(t)_1 \ \mathbf{of} \ \{\ell_1 x_1 \rightarrow \vec{\mathcal{D}}_{\bar{\Gamma}, x_1}(s_1)_2 \mid \dots \mid \ell_n x_n \rightarrow \vec{\mathcal{D}}_{\bar{\Gamma}, x_n}(s_n)_2\} \\
 \vec{\mathcal{D}}_{\bar{\Gamma}}(\mathbf{roll} \ t)_2 &\stackrel{\text{def}}{=} \mathbf{roll} \ \vec{\mathcal{D}}_{\bar{\Gamma}}(t)_2 \\
 \vec{\mathcal{D}}_{\bar{\Gamma}}(\mathbf{fold} \ t \ \mathbf{with} \ \rightarrow xs)_2 &\stackrel{\text{def}}{=} \mathbf{fold} \ \vec{\mathcal{D}}_{\bar{\Gamma}}(t)_2 \ \mathbf{with} \ v \rightarrow \\
 &\quad \mathbf{let} \ x = \mathbf{fold} \ \vec{\mathcal{D}}_{\bar{\Gamma}}(t)_1 \ \mathbf{with} \ x \rightarrow \vec{\mathcal{D}}(\tau)_1 [x^{\text{tr}} \vec{\mathcal{D}}_x(s)_1 / a] \ \mathbf{in} \ \vec{\mathcal{D}}_x(s)_2 \\
 \vec{\mathcal{D}}_{\bar{\Gamma}}(\mathbf{gen} \ \mathbf{from} \ t \ \mathbf{with} \ x \rightarrow s)_2 &\stackrel{\text{def}}{=} \mathbf{gen} \ \mathbf{from} \ \vec{\mathcal{D}}_{\bar{\Gamma}}(t)_2 \ \mathbf{with} \ v \rightarrow \mathbf{let} \ x = \vec{\mathcal{D}}_{\bar{\Gamma}}(t)_1 \ \mathbf{in} \ \vec{\mathcal{D}}_x(s)_2 \\
 \vec{\mathcal{D}}_{\bar{\Gamma}}(\mathbf{unroll} \ t)_2 &\stackrel{\text{def}}{=} \mathbf{unroll} \ \vec{\mathcal{D}}_{\bar{\Gamma}}(t)_2
 \end{aligned}$$

B.2 Reverse-mode AD

$$\begin{aligned}
 \overleftarrow{\mathcal{D}}_{\bar{\Gamma}}(\mathbf{op}(t_1, \dots, t_k))_1 &\stackrel{\text{def}}{=} \mathbf{let} \ x_1 = \overleftarrow{\mathcal{D}}_{\bar{\Gamma}}(t_1) \ \mathbf{in} \ \dots \ \mathbf{let} \ x_k = \overleftarrow{\mathcal{D}}_{\bar{\Gamma}}(t_k) \ \mathbf{in} \ \mathbf{op}(x_1) \\
 \overleftarrow{\mathcal{D}}_{\bar{\Gamma}}(x)_1 &\stackrel{\text{def}}{=} x \\
 \overleftarrow{\mathcal{D}}_{\bar{\Gamma}}(\mathbf{let} \ x = t \ \mathbf{in} \ s)_1 &\stackrel{\text{def}}{=} \mathbf{let} \ x = \overleftarrow{\mathcal{D}}_{\bar{\Gamma}}(t)_1 \ \mathbf{in} \ \overleftarrow{\mathcal{D}}_{\bar{\Gamma}, x}(s)_1 \\
 \overleftarrow{\mathcal{D}}_{\bar{\Gamma}}(\langle \rangle)_1 &\stackrel{\text{def}}{=} \langle \rangle \\
 \overleftarrow{\mathcal{D}}_{\bar{\Gamma}}(\langle t, s \rangle)_1 &\stackrel{\text{def}}{=} \langle \overleftarrow{\mathcal{D}}_{\bar{\Gamma}}(t)_1, \overleftarrow{\mathcal{D}}_{\bar{\Gamma}}(s)_1 \rangle
 \end{aligned}$$

$$\begin{aligned}
 \overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(\mathbf{fst}(t))_1 &\stackrel{\text{def}}{=} \mathbf{fst}(\overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(t)_1) \\
 \overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(\mathbf{snd}(t))_1 &\stackrel{\text{def}}{=} \mathbf{snd}(\overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(t)_1) \\
 \overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(\lambda x.t)_1 &\stackrel{\text{def}}{=} \lambda x. \langle \overleftarrow{\mathcal{D}}_{\overline{\Gamma},x}(t)_1, \lambda v. \mathbf{snd}(\overleftarrow{\mathcal{D}}_{\overline{\Gamma},x}(t)_2) \rangle \\
 \overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(t\ s)_1 &\stackrel{\text{def}}{=} \mathbf{fst}(\overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(t)_1 \overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(s)_1) \\
 \overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(\ell t)_1 &\stackrel{\text{def}}{=} \ell(\overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(t)_1) \\
 \overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(\mathbf{case\ } t \mathbf{ of}\ \{\ell_1 x_1 \rightarrow s_1 \mid \dots \mid \ell_n x_n \rightarrow s_n\})_1 &\stackrel{\text{def}}{=} \\
 &\quad \mathbf{case}\ \ell_1 x_1 \rightarrow \overleftarrow{\mathcal{D}}_{\overline{\Gamma},x_1}(s_1)_1 \mid \dots \mid \ell_n x_n \rightarrow \overleftarrow{\mathcal{D}}_{\overline{\Gamma},x_n}(s_n)_1 \mathbf{ of}\ \overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(t)_1 \{\} \\
 \overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(\mathbf{roll}\ t)_1 &\stackrel{\text{def}}{=} \mathbf{roll}\ \overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(t)_1 \\
 \overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(\mathbf{fold}\ t \mathbf{ with}\ x \rightarrow s)_1 &\stackrel{\text{def}}{=} \mathbf{fold}\ \overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(t)_1 \mathbf{ with}\ x \rightarrow \overleftarrow{\mathcal{D}}_x(s)_1 \\
 \overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(\mathbf{gen\ from}\ t \mathbf{ with}\ x \rightarrow s)_1 &\stackrel{\text{def}}{=} \mathbf{gen\ from}\ \overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(t)_1 \mathbf{ with}\ x \rightarrow \overleftarrow{\mathcal{D}}_x(s)_1 \\
 \overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(\mathbf{unroll}\ t)_1 &\stackrel{\text{def}}{=} \mathbf{unroll}\ \overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(t)_1 \\
 \\
 \overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(\mathbf{op}(t_1, \dots, t_k))_2 &\stackrel{\text{def}}{=} \mathbf{let}\ x_1 = \overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(t_1) \mathbf{ in}\ \dots \mathbf{let}\ x_k = \overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(t_k) \mathbf{ in}\ \mathbf{let}\ v = \mathit{Dop}^t(x_1, \dots, x_k; v) \mathbf{ in} \\
 &\quad (\mathbf{let}\ v = \mathbf{proj}_1\ v \mathbf{ in}\ \overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(t_1)_2) + \dots + (\mathbf{let}\ v = \mathbf{proj}_1\ v \mathbf{ in}\ \overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(t_k)_2) \\
 \overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(x)_2 &\stackrel{\text{def}}{=} \mathbf{coproj}_{\text{id}_x(x; \overline{\Gamma})}(v) \\
 \overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(\mathbf{let}\ x = t \mathbf{ in}\ s)_2 &\stackrel{\text{def}}{=} \mathbf{let}\ x = \overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(t)_1 \mathbf{ in}\ \mathbf{let}\ v = \overleftarrow{\mathcal{D}}_{\overline{\Gamma},x}(s)_2 \mathbf{ in}\ \mathbf{fst}(v) + \mathbf{let}\ v = \mathbf{snd}(v) \mathbf{ in}\ \overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(t)_2 \\
 \overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(\langle \rangle)_2 &\stackrel{\text{def}}{=} \underline{0} \\
 \overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(\langle t, s \rangle)_2 &\stackrel{\text{def}}{=} (\mathbf{let}\ v = \mathbf{fst}(v) \mathbf{ in}\ \overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(t)_2) + (\mathbf{let}\ v = \mathbf{snd}(v) \mathbf{ in}\ \overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(s)_2) \\
 \overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(\mathbf{fst}(t))_2 &\stackrel{\text{def}}{=} \mathbf{let}\ v = \langle v, \underline{0} \rangle \mathbf{ in}\ \overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(t)_2 \\
 \overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(\mathbf{snd}(t))_2 &\stackrel{\text{def}}{=} \mathbf{let}\ v = \langle \underline{0}, v \rangle \mathbf{ in}\ \overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(t)_2 \\
 \overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(\lambda x.t)_2 &\stackrel{\text{def}}{=} \mathbf{case}\ v \mathbf{ of}\ !x \otimes v \rightarrow \mathbf{fst}(\overleftarrow{\mathcal{D}}_{\overline{\Gamma},x}(t)_2) \\
 \overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(t\ s)_2 &\stackrel{\text{def}}{=} \mathbf{let}\ x = \overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(s)_1 \mathbf{ in}\ (\mathbf{let}\ v = !x \otimes v \mathbf{ in}\ \overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(t)_2) + \\
 &\quad (\mathbf{let}\ v = (\mathbf{snd}(\overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(t)_1\ x)) \bullet v \mathbf{ in}\ \overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(s)_2) \\
 \overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(\ell t)_2 &\stackrel{\text{def}}{=} \overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(t)_2 \\
 \overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(\mathbf{case\ } t \mathbf{ of}\ \{\ell_1 x_1 \rightarrow s_1 \mid \dots \mid \ell_n x_n \rightarrow s_n\})_2 &\stackrel{\text{def}}{=} \\
 &\quad \mathbf{let}\ v = \mathbf{case}\ \overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(t)_1 \mathbf{ of}\ \{\ell_1 x_1 \rightarrow \overleftarrow{\mathcal{D}}_{\overline{\Gamma},x_1}(s_1)_2 \mid \dots \mid \ell_n x_n \rightarrow \overleftarrow{\mathcal{D}}_{\overline{\Gamma},x_n}(s_n)_2\} \mathbf{ in} \\
 &\quad \mathbf{fst}\ v + \mathbf{let}\ v = \mathbf{snd}\ v \mathbf{ in}\ \overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(t)_2 \\
 \overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(\mathbf{roll}\ t)_2 &\stackrel{\text{def}}{=} \mathbf{let}\ v = \mathbf{unroll}\ v \mathbf{ in}\ \overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(t)_2 \\
 \overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(\mathbf{fold}\ t \mathbf{ with}\ x \rightarrow s)_2 &\stackrel{\text{def}}{=} \mathbf{let}\ v = (\mathbf{gen\ from}\ v \mathbf{ with}\ v \rightarrow \\
 &\quad \mathbf{let}\ x = \mathbf{fold}\ \overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(t)_1 \mathbf{ with}\ x \rightarrow \overleftarrow{\mathcal{D}}(\tau)_1 [x^t \overleftarrow{\mathcal{D}}_x(s)_1 / \alpha] \mathbf{ in}\ \overleftarrow{\mathcal{D}}_x(s)_2) \mathbf{ in}\ \overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(t)_2
 \end{aligned}$$

$$\begin{aligned} \overleftarrow{\mathcal{D}}_{\Gamma}(\text{gen from } t \text{ with } x \rightarrow s)_2 &\stackrel{\text{def}}{=} \text{let } v = (\text{fold } v \text{ with } v \rightarrow \text{let } x = \overleftarrow{\mathcal{D}}_{\Gamma}(t)_1 \text{ in } \overleftarrow{\mathcal{D}}_x(s)_2) \text{ in } \overleftarrow{\mathcal{D}}_{\Gamma}(t)_2 \\ \overleftarrow{\mathcal{D}}_{\Gamma}(\text{unroll } t)_2 &\stackrel{\text{def}}{=} \text{let } v = \text{roll } v \text{ in } \overleftarrow{\mathcal{D}}_{\Gamma}(t)_2 \end{aligned}$$

Appendix C. A Manual Proof of AD Correctness for Simply Typed Coproducts

In many implementations of CHAD, we will not have access to dependent types. Therefore, we need to give up a bit of type safety for AD on coproducts. Here, we extend the applied, manual correctness proof of the applied CHAD implementation of Vákár and Smeding (2022, Appendix A).

For coproducts, we have the following constructs in the source language:

$$\begin{aligned} \mathbf{inl} &\in \mathbf{Syn}(\tau, \tau \sqcup \sigma) \\ \mathbf{inr} &\in \mathbf{Syn}(\sigma, \tau \sqcup \sigma) \\ [_, _] &: \mathbf{Syn}(\tau, \rho) \times \mathbf{Syn}(\sigma, \rho) \rightarrow \mathbf{Syn}(\tau \sqcup \sigma, \rho). \end{aligned}$$

C.1 Forward AD

We can define

$$\begin{aligned} \overrightarrow{\mathcal{D}}_{\tau \sqcup (\sigma)}_1 &\stackrel{\text{def}}{=} \overrightarrow{\mathcal{D}}(\tau)_1 \sqcup \overrightarrow{\mathcal{D}}(\sigma)_1 \\ \overrightarrow{\mathcal{D}}_{(\tau \sqcup \sigma)}_2 &\stackrel{\text{def}}{=} \overrightarrow{\mathcal{D}}(\tau)_2 * \overrightarrow{\mathcal{D}}(\sigma)_2 \\ \overrightarrow{\mathcal{D}}(\mathbf{inl})_1 &\stackrel{\text{def}}{=} \mathbf{inl} \\ \overrightarrow{\mathcal{D}}(\mathbf{inl})_2 &\stackrel{\text{def}}{=} \underline{\lambda}v. \langle v, 0 \rangle \\ \overrightarrow{\mathcal{D}}(\mathbf{inr})_1 &\stackrel{\text{def}}{=} \mathbf{inr} \\ \overrightarrow{\mathcal{D}}(\mathbf{inr})_2 &\stackrel{\text{def}}{=} \underline{\lambda}v. \langle 0, v \rangle \\ \overrightarrow{\mathcal{D}}([t, s])_1 &\stackrel{\text{def}}{=} x \vdash \text{case } x \text{ of } \{\mathbf{inl } x \rightarrow \overrightarrow{\mathcal{D}}(t)_1 \mid x \rightarrow \overrightarrow{\mathcal{D}}(s)_1\} \\ \overrightarrow{\mathcal{D}}([t, s])_2 &\stackrel{\text{def}}{=} x \vdash \text{case } x \text{ of } \{\mathbf{inr } x \rightarrow \underline{\lambda}v. \overrightarrow{\mathcal{D}}(t)_2 \bullet (\text{fst } v) \mid x \rightarrow \underline{\lambda}v. \overrightarrow{\mathcal{D}}(s)_2 \bullet (\text{snd } v)\}. \end{aligned}$$

Then, we have that

$$\begin{aligned} \overrightarrow{\mathcal{D}}(\mathbf{inl})_1 &\in \mathbf{CSyn}(\overrightarrow{\mathcal{D}}(\tau)_1, \overrightarrow{\mathcal{D}}(\tau)_1 \sqcup \overrightarrow{\mathcal{D}}(\tau)_2) \\ \overrightarrow{\mathcal{D}}(\mathbf{inl})_2 &\in \mathbf{CSyn}(\overrightarrow{\mathcal{D}}(\tau)_1, \overrightarrow{\mathcal{D}}(\tau)_2 \multimap \overrightarrow{\mathcal{D}}(\tau)_2 * \overrightarrow{\mathcal{D}}(\sigma)_2) \\ \overrightarrow{\mathcal{D}}(\mathbf{inr})_1 &\in \mathbf{CSyn}(\overrightarrow{\mathcal{D}}(\sigma)_1, \overrightarrow{\mathcal{D}}(\tau)_1 \sqcup \overrightarrow{\mathcal{D}}(\tau)_2) \\ \overrightarrow{\mathcal{D}}(\mathbf{inr})_2 &\in \mathbf{CSyn}(\overrightarrow{\mathcal{D}}(\sigma)_1, \overrightarrow{\mathcal{D}}(\sigma)_2 \multimap \overrightarrow{\mathcal{D}}(\tau)_2 * \overrightarrow{\mathcal{D}}(\sigma)_2) \\ \overrightarrow{\mathcal{D}}([t, s])_1 &\in \mathbf{CSyn}(\overrightarrow{\mathcal{D}}(\tau)_1 \sqcup \overrightarrow{\mathcal{D}}(\sigma)_1, \overrightarrow{\mathcal{D}}(\rho)_1) \\ \overrightarrow{\mathcal{D}}([t, s])_2 &\in \mathbf{CSyn}(\overrightarrow{\mathcal{D}}(\tau)_1 \sqcup \overrightarrow{\mathcal{D}}(\sigma)_1, \overrightarrow{\mathcal{D}}(\tau)_2 * \overrightarrow{\mathcal{D}}(\sigma)_2 \multimap \overrightarrow{\mathcal{D}}(\rho)_2). \end{aligned}$$

Then, we define the following semantics:

$$\begin{aligned} \llbracket \overrightarrow{\mathcal{D}}(\tau \sqcup \sigma)_1 \rrbracket &\stackrel{\text{def}}{=} \llbracket \overrightarrow{\mathcal{D}}(\tau)_1 \rrbracket \sqcup \llbracket \overrightarrow{\mathcal{D}}(\sigma)_1 \rrbracket \\ \llbracket \overrightarrow{\mathcal{D}}(\tau \sqcup \sigma)_2 \rrbracket &\stackrel{\text{def}}{=} \llbracket \overrightarrow{\mathcal{D}}(\tau)_2 \rrbracket \times \llbracket \overrightarrow{\mathcal{D}}(\sigma)_2 \rrbracket \\ \llbracket \overrightarrow{\mathcal{D}}(\mathbf{inl})_1 \rrbracket &\stackrel{\text{def}}{=} \iota_1 \end{aligned}$$

$$\begin{aligned} \llbracket \vec{\mathcal{D}}(\mathbf{inl})_2 \rrbracket &\stackrel{\text{def}}{=} _ \mapsto x \mapsto (x, 0) \\ \llbracket \vec{\mathcal{D}}(\mathbf{inr})_1 \rrbracket &\stackrel{\text{def}}{=} \iota_2 \\ \llbracket \vec{\mathcal{D}}(\mathbf{inr})_2 \rrbracket &\stackrel{\text{def}}{=} _ \mapsto y \mapsto (0, y) \\ \llbracket \vec{\mathcal{D}}([t, s])_1 \rrbracket &\stackrel{\text{def}}{=} \llbracket \vec{\mathcal{D}}(t)_1 \rrbracket, \llbracket \vec{\mathcal{D}}(s)_1 \rrbracket \\ \llbracket \vec{\mathcal{D}}([t, s])_2 \rrbracket &\stackrel{\text{def}}{=} [x \mapsto (x', _) \mapsto \llbracket \vec{\mathcal{D}}(t)_2 \rrbracket(x)(x'), y \mapsto (y', _) \mapsto \llbracket \vec{\mathcal{D}}(t)_2 \rrbracket(y)(y')]. \end{aligned}$$

We define the forward AD logical relation $P_{\tau \sqcup \sigma}$ for coproducts on

$$(\mathbb{R} \rightarrow (\llbracket \tau \rrbracket \sqcup \llbracket \sigma \rrbracket)) \times ((\mathbb{R} \rightarrow (\llbracket \vec{\mathcal{D}}(\tau)_1 \rrbracket \sqcup \llbracket \vec{\mathcal{D}}(\sigma)_1 \rrbracket)) \times (\mathbb{R} \rightarrow \mathbb{R} \multimap (\llbracket \vec{\mathcal{D}}(\tau)_2 \rrbracket \times \llbracket \vec{\mathcal{D}}(\sigma)_2 \rrbracket)))$$

as

$$\begin{aligned} &\{(t_1 \circ f', (t_1 \circ g', x \mapsto x' \mapsto (h(x)(x'), 0))) \mid (f', (g', h')) \in P_\tau\} \cup \\ &\{(t_2 \circ f', (t_2 \circ g', x \mapsto x' \mapsto (0, h(x)(x')))) \mid (f', (g', h')) \in P_\sigma\}. \end{aligned}$$

Then, clearly, **inl** and **inr** respect this relation (almost by definition). We verify that $[t, s]$ also respects the relation provided that t and s do. Suppose that $(f, (g, h)) \in P_{\tau \sqcup \sigma}$ and $(\llbracket [t], \llbracket \vec{\mathcal{D}}(t)_1 \rrbracket, \llbracket \vec{\mathcal{D}}(t)_2 \rrbracket \rrbracket) \in P_\tau$ and $(\llbracket [s], \llbracket \vec{\mathcal{D}}(s)_1 \rrbracket, \llbracket \vec{\mathcal{D}}(s)_2 \rrbracket \rrbracket) \in P_\sigma$. We have to show that

$$\begin{aligned} &(\llbracket [t], [s] \rrbracket) \circ f, \\ &(\llbracket \llbracket \vec{\mathcal{D}}(t)_1 \rrbracket, \llbracket \vec{\mathcal{D}}(s)_1 \rrbracket \rrbracket) \circ g, \\ &z \mapsto z' \mapsto [x \mapsto (x', _) \mapsto \llbracket \vec{\mathcal{D}}(t)_2 \rrbracket(x)(x'), \\ &\quad y \mapsto (y', _) \mapsto \llbracket \vec{\mathcal{D}}(t)_2 \rrbracket(y)(y')](g(z))(h(z)(z')))) \in P_{\llbracket \rho \rrbracket}. \end{aligned}$$

Now, we have two cases:

- $(f, (g, h)) = (t_1 \circ f', (t_1 \circ g', x \mapsto x' \mapsto (h'(x)(x'), 0)))$, for $(f', (g', h')) \in P_\tau$. Then,

$$\begin{aligned} &(\llbracket [t], [s] \rrbracket) \circ f, \\ &(\llbracket \llbracket \vec{\mathcal{D}}(t)_1 \rrbracket, \llbracket \vec{\mathcal{D}}(s)_1 \rrbracket \rrbracket) \circ g, \\ &z \mapsto z' \mapsto [x \mapsto (x', _) \mapsto \llbracket \vec{\mathcal{D}}(t)_2 \rrbracket(x)(x'), \\ &\quad y \mapsto (y', _) \mapsto \llbracket \vec{\mathcal{D}}(t)_2 \rrbracket(y)(y')](g(z))(h(z)(z')))) = \\ &(\llbracket [t] \rrbracket \circ f', (\llbracket \llbracket \vec{\mathcal{D}}(t)_1 \rrbracket \rrbracket \circ g', z \mapsto z' \mapsto \llbracket \vec{\mathcal{D}}(t)_2 \rrbracket(g(z))(h(z)(z')))), \end{aligned}$$

which is a member of P_ρ because t respects the logical relation by assumption.

- $(f, (g, h)) = (t_2 \circ f', (t_2 \circ g', x \mapsto x' \mapsto (0, h'(x)(x'))))$ for $(f', (g', h')) \in P_\sigma$. Then,

$$\begin{aligned} &(\llbracket [t], [s] \rrbracket) \circ f, \\ &(\llbracket \llbracket \vec{\mathcal{D}}(t)_1 \rrbracket, \llbracket \vec{\mathcal{D}}(s)_1 \rrbracket \rrbracket) \circ g, \\ &z \mapsto z' \mapsto [x \mapsto (x', _) \mapsto \llbracket \vec{\mathcal{D}}(t)_2 \rrbracket(x)(x'), \\ &\quad y \mapsto (y', _) \mapsto \llbracket \vec{\mathcal{D}}(t)_2 \rrbracket(y)(y')](g'(z))(h'(z)(z')))) = \\ &(\llbracket [s] \rrbracket \circ f', (\llbracket \llbracket \vec{\mathcal{D}}(s)_1 \rrbracket \rrbracket \circ g', z \mapsto z' \mapsto \llbracket \vec{\mathcal{D}}(t)_2 \rrbracket(g'(z))(h'(z)(z')))), \end{aligned}$$

which is a member of P_ρ because s respects the logical relation by assumption.

It follows that our implementation of forward AD for coproducts is correct.

C.2 Reverse AD

We can define

$$\begin{aligned}
 \overleftarrow{\mathcal{D}}(\tau \sqcup \sigma)_1 &\stackrel{\text{def}}{=} \overleftarrow{\mathcal{D}}(\tau)_1 \sqcup \overleftarrow{\mathcal{D}}(\sigma)_1 \\
 \overleftarrow{\mathcal{D}}(\tau \sqcup \sigma)_2 &\stackrel{\text{def}}{=} \overleftarrow{\mathcal{D}}(\tau)_1 * \overleftarrow{\mathcal{D}}(\sigma)_1 \\
 \overleftarrow{\mathcal{D}}(\mathbf{inl})_1 &\stackrel{\text{def}}{=} \mathbf{inl} \\
 \overleftarrow{\mathcal{D}}(\mathbf{inl})_2 &\stackrel{\text{def}}{=} \underline{\lambda}v. \mathbf{fst} \ v \\
 \overleftarrow{\mathcal{D}}(\mathbf{inr})_1 &\stackrel{\text{def}}{=} \mathbf{inr} \\
 \overleftarrow{\mathcal{D}}(\mathbf{inr})_2 &\stackrel{\text{def}}{=} \underline{\lambda}v. \mathbf{snd} \ v \\
 \overleftarrow{\mathcal{D}}([t, s]_1) &\stackrel{\text{def}}{=} x \vdash \mathbf{case} \ x \ \mathbf{of} \ \{\mathbf{inl} \ x \rightarrow \overleftarrow{\mathcal{D}}(t)_1 \mid x \rightarrow \overleftarrow{\mathcal{D}}(s)_1\} \\
 \overleftarrow{\mathcal{D}}([t, s]_2) &\stackrel{\text{def}}{=} x \vdash \mathbf{case} \ x \ \mathbf{of} \ \{\mathbf{inr} \ x \rightarrow \underline{\lambda}v. \langle \overleftarrow{\mathcal{D}}(t)_2 \bullet v, 0 \rangle \mid x \rightarrow \underline{\lambda}v. \langle 0, \overleftarrow{\mathcal{D}}(s)_2 \bullet v \rangle\}
 \end{aligned}$$

Then, we have that

$$\begin{aligned}
 \overleftarrow{\mathcal{D}}(\mathbf{inl})_1 &\in \mathbf{CSyn}(\overleftarrow{\mathcal{D}}(\tau)_1, \overleftarrow{\mathcal{D}}(\tau)_1 \sqcup \overleftarrow{\mathcal{D}}(\tau)_2) \\
 \overleftarrow{\mathcal{D}}(\mathbf{inl})_2 &\in \mathbf{CSyn}(\overleftarrow{\mathcal{D}}(\tau)_1, \overleftarrow{\mathcal{D}}(\tau)_2 * \overleftarrow{\mathcal{D}}(\sigma)_2 \multimap \overleftarrow{\mathcal{D}}(\tau)_2) \\
 \overleftarrow{\mathcal{D}}(\mathbf{inr})_1 &\in \mathbf{CSyn}(\overleftarrow{\mathcal{D}}(\sigma)_1, \overleftarrow{\mathcal{D}}(\tau)_1 \sqcup \overleftarrow{\mathcal{D}}(\tau)_2) \\
 \overleftarrow{\mathcal{D}}(\mathbf{inr})_2 &\in \mathbf{CSyn}(\overleftarrow{\mathcal{D}}(\sigma)_1, \overleftarrow{\mathcal{D}}(\tau)_2 * \overleftarrow{\mathcal{D}}(\sigma)_2 \multimap \overleftarrow{\mathcal{D}}(\sigma)_2) \\
 \overleftarrow{\mathcal{D}}([t, s]_1) &\in \mathbf{CSyn}(\overleftarrow{\mathcal{D}}(\tau)_1 \sqcup \overleftarrow{\mathcal{D}}(\sigma)_1, \overleftarrow{\mathcal{D}}(\rho)_1) \\
 \overleftarrow{\mathcal{D}}([t, s]_2) &\in \mathbf{CSyn}(\overleftarrow{\mathcal{D}}(\tau)_1 \sqcup \overleftarrow{\mathcal{D}}(\sigma)_1, \overleftarrow{\mathcal{D}}(\rho)_2 \multimap \overleftarrow{\mathcal{D}}(\tau)_2 * \overleftarrow{\mathcal{D}}(\sigma)_2).
 \end{aligned}$$

Then,

$$\begin{aligned}
 \llbracket \overleftarrow{\mathcal{D}}(\tau \sqcup (\sigma))_1 \rrbracket &\stackrel{\text{def}}{=} \llbracket \overleftarrow{\mathcal{D}}(\tau)_1 \rrbracket \sqcup \llbracket \overleftarrow{\mathcal{D}}(\tau)_1 \rrbracket \\
 \llbracket \overleftarrow{\mathcal{D}}(\tau \sqcup \sigma)_2 \rrbracket &\stackrel{\text{def}}{=} \llbracket \overleftarrow{\mathcal{D}}(\tau)_2 \rrbracket \times \llbracket \overleftarrow{\mathcal{D}}(\tau)_2 \rrbracket \\
 \llbracket \overleftarrow{\mathcal{D}}(\mathbf{inl})_1 \rrbracket &\stackrel{\text{def}}{=} \iota_1 \\
 \llbracket \overleftarrow{\mathcal{D}}(\mathbf{inl})_2 \rrbracket &\stackrel{\text{def}}{=} _ \mapsto (x, _) \mapsto x \\
 \llbracket \overleftarrow{\mathcal{D}}(\mathbf{inr})_1 \rrbracket &\stackrel{\text{def}}{=} \iota_2 \\
 \llbracket \overleftarrow{\mathcal{D}}(\mathbf{inr})_2 \rrbracket &\stackrel{\text{def}}{=} _ \mapsto (_, y) \mapsto y \\
 \llbracket \overleftarrow{\mathcal{D}}([t, s]_1) \rrbracket &\stackrel{\text{def}}{=} \llbracket \llbracket \overleftarrow{\mathcal{D}}(t)_1 \rrbracket, \llbracket \overleftarrow{\mathcal{D}}(s)_1 \rrbracket \rrbracket \\
 \llbracket \overleftarrow{\mathcal{D}}([t, s]_2) \rrbracket &\stackrel{\text{def}}{=} [x \mapsto z' \mapsto (\llbracket \overleftarrow{\mathcal{D}}(t)_2 \rrbracket(x)(z'), 0), y \mapsto z' \mapsto (0, \llbracket \overleftarrow{\mathcal{D}}(t)_2 \rrbracket(y)(z'))].
 \end{aligned}$$

We define the reverse AD logical relation $P_{\tau\text{sigma}}$ for coproducts on

$$(\mathbb{R} \rightarrow (\llbracket \tau \rrbracket \sqcup \llbracket \sigma \rrbracket)) \times ((\mathbb{R} \rightarrow (\llbracket \overleftarrow{\mathcal{D}}(\tau)_1 \rrbracket \sqcup \llbracket \overleftarrow{\mathcal{D}}(\sigma)_1 \rrbracket)) \times (\mathbb{R} \rightarrow (\llbracket \overleftarrow{\mathcal{D}}(\tau)_2 \rrbracket \times \llbracket \overleftarrow{\mathcal{D}}(\sigma)_2 \rrbracket) \multimap \mathbb{R}))$$

as

$$\begin{aligned}
 &\{(\iota_1 \circ f', (\iota_1 \circ g', z \mapsto (x', _) \mapsto h'(z)(x'))) \mid (f', (g', h')) \in P_\tau\} \cup \\
 &\{(\iota_2 \circ f', (\iota_2 \circ g', z \mapsto (_, y') \mapsto h'(z)(y'))) \mid (f', (g', h')) \in P_\sigma\}.
 \end{aligned}$$

Then, clearly, **inl** and **inr** respect this relation (almost by definition). We verify that $[t, s]$ also respects the relation provided that t and s do. Suppose that $(f, (g, h)) \in P_{\tau \sqcup \sigma}$ and $(\llbracket t \rrbracket, (\llbracket \overleftarrow{\mathcal{D}}(t)_1 \rrbracket, \llbracket \overleftarrow{\mathcal{D}}(t)_2 \rrbracket)) \in P_\tau$ and $(\llbracket s \rrbracket, (\llbracket \overleftarrow{\mathcal{D}}(s)_1 \rrbracket, \llbracket \overleftarrow{\mathcal{D}}(s)_2 \rrbracket)) \in P_\sigma$. We have to show that

$$\begin{aligned} & (\llbracket t \rrbracket, \llbracket s \rrbracket) \circ f, \\ & (\llbracket \overleftarrow{\mathcal{D}}(t)_1 \rrbracket, \llbracket \overleftarrow{\mathcal{D}}(s)_1 \rrbracket) \circ g, \\ & z \mapsto x' \mapsto h(z)([x \mapsto z' \mapsto (\llbracket \overleftarrow{\mathcal{D}}(t)_2 \rrbracket(x)(z'), 0), \\ & \quad y \mapsto z' \mapsto (0, \llbracket \overleftarrow{\mathcal{D}}(s)_2 \rrbracket(y)(z'))](g(x))(x')))) \in P_{\llbracket \rho \rrbracket}. \end{aligned}$$

Now, we have two cases:

- $(f, (g, h)) = (\iota_1 \circ f', (\iota_1 \circ g', z \mapsto (x', _) \mapsto h'(z)(x')))$, for $(f', (g', h')) \in P_\tau$. Then,

$$\begin{aligned} & (\llbracket t \rrbracket, \llbracket s \rrbracket) \circ f, \\ & (\llbracket \overleftarrow{\mathcal{D}}(t)_1 \rrbracket, \llbracket \overleftarrow{\mathcal{D}}(s)_1 \rrbracket) \circ g, \\ & z \mapsto x' \mapsto h(z)([x \mapsto z' \mapsto (\llbracket \overleftarrow{\mathcal{D}}(t)_2 \rrbracket(x)(z'), 0), \\ & \quad y \mapsto z' \mapsto (0, \llbracket \overleftarrow{\mathcal{D}}(s)_2 \rrbracket(y)(z'))](g(x))(x')))) = \\ & (\llbracket t \rrbracket \circ f', (\llbracket \overleftarrow{\mathcal{D}}(t)_1 \rrbracket \circ g', z \mapsto x' \mapsto h'(z)(\llbracket \overleftarrow{\mathcal{D}}(t)_2 \rrbracket(g'(x))(x')))), \end{aligned}$$

which is a member of P_ρ because t respects the logical relation by assumption:

- $(f, (g, h)) = (\iota_2 \circ f', (\iota_2 \circ g', z \mapsto (_, y') \mapsto h'(z)(y')))$ for $(f', (g', h')) \in P_\sigma$. Then,

$$\begin{aligned} & (\llbracket t \rrbracket, \llbracket s \rrbracket) \circ f, \\ & (\llbracket \overleftarrow{\mathcal{D}}(t)_1 \rrbracket, \llbracket \overleftarrow{\mathcal{D}}(s)_1 \rrbracket) \circ g, \\ & z \mapsto x' \mapsto h(z)([x \mapsto z' \mapsto (\llbracket \overleftarrow{\mathcal{D}}(t)_2 \rrbracket(x)(z'), 0), \\ & \quad y \mapsto z' \mapsto (0, \llbracket \overleftarrow{\mathcal{D}}(s)_2 \rrbracket(y)(z'))](g(x))(x')))) = \\ & (\llbracket s \rrbracket \circ f', (\llbracket \overleftarrow{\mathcal{D}}(s)_1 \rrbracket \circ g', z \mapsto x' \mapsto h'(z)(\llbracket \overleftarrow{\mathcal{D}}(s)_2 \rrbracket(g'(x))(x')))), \end{aligned}$$

which is a member of P_ρ because s respects the logical relation by assumption.

It follows that our implementation of reverse AD for coproducts is correct.

A categorical way to understand this proof is that $(A_1, A_2) \sqcup (B_1, B_2) \stackrel{\text{def}}{=} (A_1 \sqcup B_1, A_2 \times B_2)$ lifts the coproduct in \mathcal{C} to a *weak* (fibered) coproduct in $\Sigma_{\mathcal{C}}\mathcal{L}$ and $\Sigma_{\mathcal{C}}\mathcal{L}^{op}$. This weak coproduct lifts to the subscone, in the manner outlined above. One consequence is that the AD transformations no longer respect the η -rule for coproducts (unlike in the dependently typed setting).