Pré-Publicações do Departamento de Matemática
Universidade de Coimbra
Preprint Number 21–36

CHAD FOR EXPRESSIVE TOTAL LANGUAGES

FERNANDO LUCATELLI NUNES AND MATTHIJS VÁKÁR

Abstract: We show how to apply forward and reverse mode Combinatory Homomorphic Automatic Differentiation (CHAD) 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 analysing 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, program correctness, denotational semantics, variant types, inductive types, coinductive types, extensive indexed categories, Artin glueing, logical relations.
Math. Subject Classification (2020): 68Nxx, 68Txx, 68Wxx, 18Cxx, 18A25, 18D30, 18B50.

Introduction

Automatic differentiation (AD) is a popular technique for computing derivatives of functions implemented by computer programs, essentially by applying the chain-rule across 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,

Received October 04, 2021.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No. 895827. This research was partially supported by the Centre for Mathematics of the University of Coimbra - UIDB/00324/2020, funded by the Portuguese Government through FCT/MCTES.

for a function $f : \mathbb{R}^n \to \mathbb{R}^m$, 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 (gradients) of functions $f : \mathbb{R}^n \to \mathbb{R}$ with very large n, reverse AD tends to be the more important algorithm in practice.

While the study of AD has a long history in the numerical methods community, which we will not survey (see, for example, [16]), 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 10. 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 Cor. 9.1). Specifically, we extend the scope of the Combinatory Homomorphic Automatic Differentiation (CHAD) method of forward and reverse AD [40, 39] (from the previous state of the art: a simply typed λ-calculus) to apply to total functional programming languages with expressive type systems, i.e. 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 pay-off of this challenging development are a surprisingly simple AD algorithm as well as reusable abstract semantic techniques.

The main contributions of this paper are:

- developing an abstract categorical semantics (§1) of such expressive type systems in suitable Σ-types of categories (§3);
- presenting, as the initial instantiation of this abstract semantics, an idealised target language for CHAD when applied to such type systems (§4);
- deriving the forward and reverse CHAD algorithms when applied to expressive type systems as the uniquely defined homomorphic functors (§5) from the source (§2) to the target language (§4);
- introducing (categorical) logical relations techniques (aka sconing) for reasoning about expressive functional languages that include both inductive and coinductive types (§7)
- using such a logical relations construction over the concrete denotational semantics of the source and target languages (§6) that demonstrates that CHAD correctly calculates the usual mathematical derivative (§8);
- discussing applied considerations around implementing this extended CHAD method in practice (§9).

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

Key ideas

Origins in semantic derivatives and chain rules. CHAD starts from the observation that for a smooth function

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

it is useful to pair the primal function value $f(x)$ with f's derivative $Df(x)$ at x (where we underline the spaces $\underline{\mathbb{R}}^n$ 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):

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

if we want to calculate derivatives in a compositional way. 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 $\mathcal{T}_i f \stackrel{\text{def}}{=} \pi_i \circ \mathcal{T}f$, for $i = 1, 2$:

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

We make two observations:

(1) the derivative of $g \circ f$ does not only depend 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

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

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.

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

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

such as constants (as nullary operations), (elementwise) addition and multiplication of arrays, inner products and certain non-linear 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

$$\begin{aligned}
&\mathbf{let}\, y = \ x_1 * x_4 + 2 * x_2 \,\mathbf{in} \\
&\mathbf{let}\, z = \ y * x_3 \,\mathbf{in} \\
&\mathbf{let}\, w = \ z + x_4 \,\mathbf{in}\, \sin(w),
\end{aligned}$$

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

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 linear types that are equipped with a commutative monoid structure $(\underline{0}, +)$. Indeed, (transposed) derivatives are linear functions: homomorphisms of this monoid structure. We extend these operations $\overrightarrow{\mathcal{D}}$ and $\overleftarrow{\mathcal{D}}$ to act on typing contexts Γ:

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

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

$$\begin{aligned}
\overrightarrow{\mathcal{D}}(\Gamma)_1 &\vdash \overrightarrow{\mathcal{D}}_{\overline{\Gamma}}(t) : \overrightarrow{\mathcal{D}}(\sigma) * (\overrightarrow{\mathcal{D}}(\Gamma)_2 \multimap \overrightarrow{\mathcal{D}}(\sigma)) \\
\overleftarrow{\mathcal{D}}(\Gamma)_1 &\vdash \overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(t) : \overleftarrow{\mathcal{D}}(\sigma) * (\overleftarrow{\mathcal{D}}(\Gamma)_2 \multimap \overleftarrow{\mathcal{D}}(\sigma)).
\end{aligned}$$

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

Theorem A (Correctness of CHAD, Thm. 8.1)**.** For any well-typed program

$$x_1 : \mathbf{real}^{n_1}, \ldots, x_k : \mathbf{real}^{n_k} \vdash t : \mathbf{real}^m$$

we have that $[\![\overrightarrow{\mathcal{D}}(t)]\!] = \mathcal{T}[\![t]\!]$ and $[\![\overleftarrow{\mathcal{D}}(t)]\!] = \mathcal{T}^*[\![t]\!]$.

Once we fix a 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:

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

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

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

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

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

Here, we write $\underline{\lambda}\mathsf{v}.t$ for a linear function abstraction (merely a notational convention – it can simply be though 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). Further, given $\Gamma; \mathsf{v} : \underline{\tau} \vdash t : (\underline{\sigma}_1 \boldsymbol{*} \cdots \boldsymbol{*} \underline{\sigma}_n)$, we write $\Gamma; \mathsf{v} : \underline{\tau} \vdash \mathbf{proj}_i\,(t) : \underline{\sigma}_i$ for the i-th projection of t. Similarly, given $\Gamma; \mathsf{v} : \underline{\tau} \vdash t : \underline{\sigma}_i$, we write the i-th coprojection $\Gamma; \mathsf{v} : \underline{\tau} \vdash \mathbf{coproj}_i\,(t) = \langle \underline{0}, \ldots, \underline{0}, t, \underline{0}, \ldots, \underline{0}\rangle : (\underline{\sigma}_1 \boldsymbol{*} \cdots \boldsymbol{*} \underline{\sigma}_n)$ and we write $\mathbf{idx}(x_i; x_1, \ldots, x_n) = i$ for the index of an identifier in a list of identifiers. Finally, $D\mathrm{op}^t$ 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 $D\mathrm{op}^t$ of the basic building blocks op.

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 $\mathbb{R}^n \to \mathbb{R}^m$ and 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.

In [40, 39], we 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 $\mathbf{LSyn} : \mathbf{CSyn}^{op} \to \mathbf{Cat}$, i.e. functor to the category of categories and functors for which $\mathrm{ob}\,\mathbf{LSyn}(\tau) = \mathrm{ob}\,\mathbf{LSyn}(\sigma)$ for all $\tau, \sigma \in \mathrm{ob}\,\mathbf{CSyn}$ and

$$\mathbf{LSyn}(\tau \xrightarrow{t} \sigma) : \mathbf{LSyn}(\sigma) \to \mathbf{LSyn}(\tau)$$

is identity on objects. Here, $\mathbf{CSyn}$ is the syntactic category whose objects are Cartesian types τ, σ, ρ and morphisms $\tau \to \sigma$ 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 $\underline{\tau}, \underline{\sigma}, \underline{\rho}$ and morphisms $\underline{\sigma} \to \underline{\rho}$ are programs $x : \tau; \mathsf{v} : \underline{\sigma} \vdash t : \underline{\rho}$ of type $\underline{\rho}$ that have a free variable x of Cartesian type τ and a free variable v of linear type $\underline{\sigma}$. The key observation then is the following.

Theorem B (CHAD from a universal property, Cor. 5.1)**.** Forward and reverse mode CHAD are the unique structure preserving functors

$$\overrightarrow{\mathcal{D}}(-) : \mathbf{Syn} \to \Sigma_{\mathbf{CSyn}}\mathbf{LSyn}$$
$$\overleftarrow{\mathcal{D}}(-) : \mathbf{Syn} \to \Sigma_{\mathbf{CSyn}}\mathbf{LSyn}^{op}$$

from the syntactic category $\mathbf{Syn}$ of the source language to (opposite) Grothendieck construction of the target language $\mathbf{LSyn} : \mathbf{CSyn}^{op} \to \mathbf{Cat}$ that send primitive operations op to their derivative Dop and transposed derivative $D\mathrm{op}^t$, 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 [40, 39] to the idea to try to use Thm. B as a definition of CHAD on more expressive programming languages. In particular, they consider a higher-order functional source language $\mathbf{Syn}$, i.e. the freely generated Cartesian closed category on the objects $\mathbf{real}^n$ and morphisms op and try to define $\overrightarrow{\mathcal{D}}(-)$ and $\overleftarrow{\mathcal{D}}(-)$ 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 $\mathbf{LSyn} : \mathbf{CSyn}^{op} \to \mathbf{Cat}$. The main contribution then is to identify conditions on a locally indexed category $\mathcal{L} : \mathcal{C}^{op} \to \mathbf{Cat}$ that guarantee that $\Sigma_{\mathcal{C}}\mathcal{L}$ and $\Sigma_{\mathcal{C}}\mathcal{L}^{op}$ are Cartesian closed and to take the target language $\mathbf{LSyn} : \mathbf{CSyn}^{op} \to \mathbf{Cat}$ 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 $\Sigma_{\mathcal{C}}\mathcal{L}$ and $\Sigma_{\mathcal{C}}\mathcal{L}^{op}$. 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.

Further, we observe in these papers that Thm. 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 sconing) that relates smooth curves to their associated primal and (co)tangent curves.

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 analyse 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, [41] 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.

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 Thm. 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 $\mathcal{L} : \mathcal{C}^{op} \to \mathbf{Cat}$ under which $\Sigma_{\mathcal{C}}\mathcal{L}$ and $\Sigma_{\mathcal{C}}\mathcal{L}^{op}$ have distributive coproducts. A 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 $\mathcal{L} : \mathcal{C}^{op} \to \mathbf{Cat}$.

In fact, this is compatible with what we know from differential geometry: coproducts allow us to construct spaces with multiple connected components, each of which may have a distinct dimension. To make things concrete: the tangent space $\mathcal{T}_x(\mathbb{R}^2 + \mathbb{R}^3)$ is either $\underline{\mathbb{R}}^2$ or $\underline{\mathbb{R}}^3$ depending on whether the base point x is chosen in the left or right component of the coproduct. 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 $\vdash \overrightarrow{\mathcal{D}}(\tau)_1 : \mathrm{type}$ and $\vdash \overleftarrow{\mathcal{D}}(\tau)_1 : \mathrm{type}$ can remain (closed/non-dependent) Cartesian types, $p : \overrightarrow{\mathcal{D}}(\tau)_1 \vdash \overrightarrow{\mathcal{D}}(\tau)_2 : \mathrm{ltype}$ and $p : \overleftarrow{\mathcal{D}}(\tau)_1 \vdash \overleftarrow{\mathcal{D}}(\tau)_2 : \mathrm{ltype}$ 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 $\mathcal{L} : \mathcal{C}^{op} \to \mathbf{Cat}$ instead of merely locally indexed ones.

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

$$\begin{array}{c}\overrightarrow{\mathcal{D}}(\Gamma)_1 \vdash \overrightarrow{\mathcal{D}}_{\overline{\Gamma}}(t) : \Sigma p : \overrightarrow{\mathcal{D}}(\tau)_1 . \overrightarrow{\mathcal{D}}(\Gamma)_2 \multimap \overrightarrow{\mathcal{D}}(\tau)_2 \\ \overleftarrow{\mathcal{D}}(\Gamma)_1 \vdash \overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(t) : \Sigma p : \overleftarrow{\mathcal{D}}(\tau)_1 . \overleftarrow{\mathcal{D}}(\tau)_2 \multimap \overleftarrow{\mathcal{D}}(\Gamma)_2,\end{array}$$

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

$$\begin{aligned}\overrightarrow{\mathcal{D}}(\Gamma)_2 &\stackrel{\text{def}}{=} (\overrightarrow{\mathcal{D}}(\tau_1)_2[{}^{x_1}/_p] * \cdots * \overrightarrow{\mathcal{D}}(\tau_n)_2[{}^{x_n}/_p]) \\ \overleftarrow{\mathcal{D}}(\Gamma)_2 &\stackrel{\text{def}}{=} (\overleftarrow{\mathcal{D}}(\tau_1)_2[{}^{x_1}/_p] * \cdots * \overleftarrow{\mathcal{D}}(\tau_n)_2[{}^{x_n}/_p]).\end{aligned}$$

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

We prove the following novel results.

Theorem C (Bi-Cartesian closed structure of Σ-categories, Prop. 3.1 and 3.2, Thm. 3.1, 3.2 and 3.4 and Cor. 3.6 and 3.7)**.** For a category $\mathcal{C}$ and a strictly indexed category $\mathcal{L} : \mathcal{C}^{op} \to \mathbf{Cat}$, $\Sigma_{\mathcal{C}}\mathcal{L}$ and $\Sigma_{\mathcal{C}}\mathcal{L}^{op}$ have

- (fibred) finite products, if $\mathcal{C}$ has finite coproducts and $\mathcal{L}$ has strictly indexed products and coproducts;
- (fibred) 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 intentially keep this vague here to aid legibility – the point is that these are relatively standard conditions).

Further the coproducts in $\Sigma_{\mathcal{C}}\mathcal{L}$ and $\Sigma_{\mathcal{C}}\mathcal{L}^{op}$ are distribute over the products, as long as those in $\mathcal{C}$ do, even in absence of exponentials. Notably, the exponentials are not generally fibred 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 $\mathcal{L} : \mathcal{C}^{op} \to \mathbf{Cat}$ *extensive* if the canonical functor $\mathcal{L}(\sqcup_{i=1}^n C_i) \to \prod_{i=1}^n \mathcal{L}(C_i)$ is an equivalence. Further, we note that we need to re-establish 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

$$\mathbf{LSyn} : \mathbf{CSyn}^{op} \to \mathbf{Cat}$$

for CHAD on a source language with sum types (and tuple and function types), derive the forward and reverse CHAD algorithms for such a language and reestablish Thms. B and A 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 Thm. A uses the universal property of Thm. B and a logical relations (categorical sconing) construction over the denotational semantics of the source and target languages. This logical relations construction is relatively straightforward and relies on well-known sconing methods for bicartesian closed categories. In particular, we obtain the following formulas for a sum type $\{\ell_1\tau_1 \mid \cdots \mid \ell_n\tau_n\}$ with constructors $\ell_1, \ldots, \ell_n$ that take arguments of type $\tau_1, \ldots, \tau_n$:

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

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

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 analyse CHAD for inductive and coinductive types. If we are 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 [36], we work with the class of *$\mu\nu$-polynomials*, a relatively standard choice: i.e. 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 $\{\ell_1\tau_1 \mid \cdots \mid \ell_n\tau_n\}$ 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-fibred nature of exponentials in $\Sigma_{\mathcal{C}}\mathcal{L}$ and $\Sigma_{\mathcal{C}}\mathcal{L}^{op}$ would significantly complicate the technical development. While this excludes certain examples like the free state monad (which, for type σ state would be the intial algebra $\mu\alpha.\{Get(\sigma \to \alpha) \mid Put(\sigma{*}\alpha)\}$), it still includes the vast majority of examples of eager and lazy types that one uses in practice: e.g., lists $\mu\alpha.\{Empty\,\mathbf{1} \mid Cons(\sigma{*}\alpha)\}$, (finitely branching) labelled 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 $\mathcal{L} : \mathcal{C}^{op} \to \mathbf{Cat}$ that guarantee that $\Sigma_{\mathcal{C}}\mathcal{L}$ and $\Sigma_{\mathcal{C}}\mathcal{L}^{op}$ 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 $\Sigma_{\mathcal{C}}\mathcal{L}$ and $\Sigma_{\mathcal{C}}\mathcal{L}^{op}$. 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, Cor. F.1 and Thm. G.2)**.** Let E be a split fibration endofunctor on $\Sigma_{\mathcal{C}}\mathcal{L}$ (resp. $\Sigma_{\mathcal{C}}\mathcal{L}^{op}$) and let $(\overline{E}, e)$ be the corresponding strictly indexed endofunctor on $\mathcal{L}$. Then, E has a (fibred) initial algebra if

- $\overline{E} : \mathcal{C} \to \mathcal{C}$ has an initial algebra $\mathfrak{in}_{\overline{E}} : E(\mu\overline{E}) \to \mu\overline{E}$;
- $\mathcal{L}(\mathfrak{in}_{\overline{E}})^{-1} e_{\mu\overline{E}} : \mathcal{L}(\mu\overline{E}) \to \mathcal{L}(\mu\overline{E})$ 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 (fibred) terminal coalgebra if

- $\overline{E} : \mathcal{C} \to \mathcal{C}$ has a terminal coalgebra $\mathfrak{out}_{\overline{E}} : \nu\overline{E} \to E(\nu\overline{E})$;
- $\mathcal{L}(\mathfrak{out}_{\overline{E}}) e_{\mu\overline{E}} : \mathcal{L}(\nu\overline{E}) \to \mathcal{L}(\nu\overline{E})$ 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 (fibred) $\mu\nu$-polynomials (including their fibred initial algebras and terminal coalgebras) to exist in $\Sigma_{\mathcal{C}}\mathcal{L}$ and $\Sigma_{\mathcal{C}}\mathcal{L}^{op}$. In particular, we show that it suffices to extend the target language $\mathbf{LSyn} : \mathbf{CSyn}^{op} \to \mathbf{Cat}$ 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 Thm. 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 $\tau.\mathbf{roll}^{-1}\,x \stackrel{\mathrm{def}}{=} \mathbf{fold}\,x\,\mathbf{with}\,y \to \tau[{}^{y\vdash\mathbf{roll}\,y}/_{\alpha}]$, where we write $\tau[{}^{y\vdash\mathbf{roll}\,y}/_{\alpha}]$ 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:

$$
\begin{aligned}
&\overrightarrow{\mathcal{D}}(\alpha)_1 \stackrel{\mathrm{def}}{=} \alpha && \overrightarrow{\mathcal{D}}(\alpha)_2 = \underline{\alpha}\\
&\overrightarrow{\mathcal{D}}(\mu\alpha.\tau)_1 \stackrel{\mathrm{def}}{=} \mu\alpha.\overrightarrow{\mathcal{D}}(\tau)_1 && \overrightarrow{\mathcal{D}}(\mu\alpha.\tau)_2 \stackrel{\mathrm{def}}{=} \underline{\mu}\underline{\alpha}.\overrightarrow{\mathcal{D}}(\tau)_2[{}^{\overrightarrow{\mathcal{D}}(\tau)_1.\mathbf{roll}^{-1}\,p}/_{p}]\\
&\overrightarrow{\mathcal{D}}(\nu\alpha.\tau)_1 \stackrel{\mathrm{def}}{=} \nu\alpha.\overrightarrow{\mathcal{D}}(\tau)_1 && \overrightarrow{\mathcal{D}}(\nu\alpha.\tau)_2 \stackrel{\mathrm{def}}{=} \underline{\nu}\underline{\alpha}.\overrightarrow{\mathcal{D}}(\tau)_2[{}^{\mathbf{unroll}\,p}/_{p}]\\
&\overleftarrow{\mathcal{D}}(\alpha)_1 \stackrel{\mathrm{def}}{=} \alpha && \overleftarrow{\mathcal{D}}(\alpha)_2 = \underline{\alpha}\\
&\overleftarrow{\mathcal{D}}(\mu\alpha.\tau)_1 \stackrel{\mathrm{def}}{=} \mu\alpha.\overleftarrow{\mathcal{D}}(\tau)_1 && \overleftarrow{\mathcal{D}}(\mu\alpha.\tau)_2 \stackrel{\mathrm{def}}{=} \underline{\nu}\underline{\alpha}.\overleftarrow{\mathcal{D}}(\tau)_2[{}^{\overleftarrow{\mathcal{D}}(\tau)_1.\mathbf{roll}^{-1}\,p}/_{p}]\\
&\overleftarrow{\mathcal{D}}(\nu\alpha.\tau)_1 \stackrel{\mathrm{def}}{=} \nu\alpha.\overleftarrow{\mathcal{D}}(\tau)_1 && \overleftarrow{\mathcal{D}}(\nu\alpha.\tau)_2 \stackrel{\mathrm{def}}{=} \underline{\mu}\underline{\alpha}.\overleftarrow{\mathcal{D}}(\tau)_2[{}^{\mathbf{unroll}\,p}/_{p}]
\end{aligned}
$$

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

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

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

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 $\mathbf{FVect} : \mathbf{Set}^{op} \to \mathbf{Cat}$ of families of vector spaces also satisfies our conditions, so it gives a concrete denotational semantics of the target language $\mathbf{LSyn} : \mathbf{CSyn}^{op} \to \mathbf{Cat}$, by Thm. 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 analysing 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, §7)**.** Let $G : \mathcal{C} \to \mathcal{D}$ be a functor. We observe

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

This is relevant because:

- comonadic functors create initial algebras;
- monadic functors create terminal coalgebras;
- monadic-comonadic functors create $\mu\nu$-polynomials;
- if $\mathcal{E}$ is monadic-comonadic over $\mathcal{E}'$, then $\mathcal{E}$ is finitely complete cartesian closed if $\mathcal{E}'$ is.

These logical relations techniques are suffient to yield the correctness theorem A. Indeed, as long as derivatives of primitive operations are correctly implemented in the sense that $[\![D\mathrm{op}]\!] = D\mathrm{op}$ and $[\![D\mathrm{op}^t]\!] = D[\![\mathrm{op}]\!]^t$, Thm. E tells us that the unique structure preserving functors

$$([\![-]\!], [\![\overrightarrow{\mathcal{D}}(-)]\!]) : \mathbf{Syn} \to \mathbf{Set} \times \Sigma_{\mathbf{Set}}\mathbf{FVect}$$
$$([\![-]\!], [\![\overleftarrow{\mathcal{D}}(-)]\!]) : \mathbf{Syn} \to \mathbf{Set} \times \Sigma_{\mathbf{Set}}\mathbf{FVect}^{op}$$

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

$$\{(f, (g, h)) \mid f = g \text{ and } h = Df\ \} \hookrightarrow (\mathbf{Set} \times \Sigma_{\mathbf{Set}}\mathbf{FVect}\quad)\,((\mathbb{R}, (\mathbb{R}, \underline{\mathbb{R}})), (\mathbb{R}^n, (\mathbb{R}^n, \underline{\mathbb{R}}^n)))$$
$$\{(f, (g, h)) \mid f = g \text{ and } h = Df^t\} \hookrightarrow (\mathbf{Set} \times \Sigma_{\mathbf{Set}}\mathbf{FVect}^{op})\,((\mathbb{R}, (\mathbb{R}, \underline{\mathbb{R}})), (\mathbb{R}^n, (\mathbb{R}^n, \underline{\mathbb{R}}^n)))\,.$$

We see that $[\![\overrightarrow{\mathcal{D}}(t)]\!]$ and $[\![\overleftarrow{\mathcal{D}}(t)]\!]$ propagate derivatives and transposed derivatives of smooth curves, correctly, for all programs t. This is sufficient as every tangent/cotangent vector to $\mathbb{R}^n$ can be represented as the derivative/transposed derivative of a smooth curve.

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.

How does CHAD for expressive types work in practice? The CHAD code transformations we describe in this papers are well-behaved in practical implementations in the sense of the following compile-time complexity result.

Theorem F (No code blow-up, Cor. 9.1)**.** 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.

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 run-time 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.

1. 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 [34, 36], which builds on the usual semantics of the simply typed λ-calculus in Cartesian closed categories [24]. 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** (§2), 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** (§6) in terms of sets and functions, which represents our default denotational semantics of the source language;
- models $\Sigma_{\mathcal{C}}\mathcal{L}$ and $\Sigma_{\mathcal{C}}\mathcal{L}^{op}$ (§3) in the the Σ-types of suitable indexed categories $\mathcal{L} : \mathcal{C}^{op} \to \mathbf{Cat}$;
- in particular, the models $\Sigma_{\mathbf{CSyn}}\mathbf{LSyn}$ and $\Sigma_{\mathbf{CSyn}}\mathbf{LSyn}^{op}$ (§4) built out of the target language, which yield forward and reverse mode CHAD code transformations, respectively;
- sconing (categorical logical relations) constructions $\overrightarrow{\mathbf{Scone}}$ and $\overleftarrow{\mathbf{Scone}}$ (§7) 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 $\mathbf{FVect} : \mathbf{Set}^{op} \to \mathbf{Cat}$ 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.

1.1. Basics. 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 $g \circ f$ (also written $f; g$) 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.

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

1.3. Primitive types and operations. Often, we are interested in programming languages that have support for a certain set Ty of basic types such as integers and (floating point) real numbers as well as certain sets $\mathsf{Op}(T_1, \ldots, T_n; S)$, for $T_1, \ldots, T_n, S \in \mathsf{Ty}$, of operations on these basic types such as addition, multiplication, sine functions, etc.. We model such primitive types and operations categorically by demanding that our category has a distinguished object C_T for each $T \in \mathsf{Ty}$ to represent the primitive types and a distinguished morphism $f_{\mathrm{op}} \in \mathcal{C}(C_{T_1} \times \ldots \times C_{T_n}, C_S)$ for all primitive operations $\mathrm{op} \in \mathsf{Op}(T_1, \ldots, T_n; S)$.

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

1.5. 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 modelled as a coproduct $\coprod_{i\in I} A_i$ of a collection of types $\{A_i\}_{i\in I}$, which we also write $\mathbb{0}$ or $A_1 \sqcup A_2$ in the case of nullary and binary coproducts. We write $[f_i]_{i\in I} : \coprod_{i\in I} C_i \to A$ for the copairing of $\{f_i : C_i \to A\}_{i\in I}$ and $\iota_j : A_j \to \coprod_{i\in I} A_i$ 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, i.e. coproducts $\coprod_{i\in I} A_i$ such that the map $[(\pi_1; \iota_i, \pi_2)]_{i\in I} : \coprod_{i\in I}(A_i \times B) \to (\coprod_{i\in I} A_i) \times B$ is an isomorphism. Note that in presence of function types, coproducts are automatically distributive. As such, we say that a categorical semantics $\mathcal{C}$ models (finite) sum types if $\mathcal{C}$ has (chosen) finite distributive coproducts.

1.6. Inductive and coinductive types. We employ the usual semantic interpretation of *inductive and coinductive types* as, respectively, *initial algebras* and *final coalgebras* of a certain class of functors. Most of this section is dedicated to describing precisely of which class of functors we consider initial algebras and final coalgebras, a class we call *$\mu\nu$-polynomials.* We briefly establish the terminology below, and refer to Appendix D for a more detailed review.

Let $E : \mathcal{D} \to \mathcal{D}$ be an endofunctor. We denote, respectively, by $(\mu E, \mathfrak{in}_E)$ and $(\nu E, \mathfrak{out}_E)$ the initial E-algebra and the final E-coalgebra, assuming their existence. Given any E-algebra (Y, ξ) and any E-coalgebra (X, ϱ), we respectively denote by

$$\mathrm{fold}_E(Y, \xi) : \mu E \to Y, \quad \mathrm{unfold}_E(X, \varrho) : X \to \nu E \tag{1.1}$$

the underlying morphisms in $\mathcal{D}$ of the unique E-algebra morphism $(\mu E, \mathfrak{in}_E) \to (Y, \xi)$ and the unique E-coalgebra morphism $(X, \varrho) \to (\nu E, \mathfrak{out}_E)$. By abuse of language, whenever it is clear from the context, we denote $\mathrm{fold}_E(Y, \xi)$ by $\mathrm{fold}_E \xi$, and $\mathrm{unfold}_E(X, \varrho)$ by $\mathrm{unfold}_E \varrho$.

In this setting, given a functor

$$H : \mathcal{D}' \times \mathcal{D} \to \mathcal{D}, \tag{1.2}$$

for each object X of $\mathcal{D}'$, we denote by H^X the endofunctor

$$H(X, -) : \mathcal{D} \to \mathcal{D}. \tag{1.3}$$

In this setting, if μH^X exists for any object $X \in \mathcal{D}'$, the initial algebras' universal properties induce a functor, denoted by $\mu H : \mathcal{D}' \to \mathcal{D}$, given by

$$\begin{array}{rcl} \mu H : \mathcal{D}' & \to & \mathcal{D} \\ X & \mapsto & \mu H^X \\ (f : X \to Y) & \mapsto & \mathrm{fold}_{H^X}\left(\mathfrak{in}_{H^Y} \circ H(f, \mu H^Y)\right). \end{array}$$

Dually, we have an induced functor $\nu H : \mathcal{D}' \to \mathcal{D}$, given by

$$\begin{array}{rcl} \nu H : \mathcal{D}' & \to & \mathcal{D} \\ X & \mapsto & \nu H^X \\ (f : X \to Y) & \mapsto & \mathrm{unfold}_{H^Y}\left(H(f, \nu H^X) \circ \mathfrak{out}_{H^X}\right), \end{array}$$

provided that the suitable final coalgebras exist. See Proposition D.1 for more details.

Definition 1.1 ($\mu\nu$-polynomials)**.** Assuming that $\mathcal{D}$ has finite coproducts and finite products, the category $\mu\nu\mathsf{Poly}_{\mathcal{D}}$ is the smallest subcategory of **Cat** satisfying the following.

– The objects are defined inductively by:
  O1. the terminal category $\mathbb{1}$ is an object of $\mu\nu\mathsf{Poly}_{\mathcal{D}}$;
  O2. the category $\mathcal{D}$ is an object of $\mu\nu\mathsf{Poly}_{\mathcal{D}}$;
  O3. for any pair of objects $(\mathcal{D}', \mathcal{D}'') \in \mu\nu\mathsf{Poly}_{\mathcal{D}} \times \mu\nu\mathsf{Poly}_{\mathcal{D}}$, the product $\mathcal{D}' \times \mathcal{D}''$ is an object of $\mu\nu\mathsf{Poly}_{\mathcal{D}}$.

– The morphisms satisfy the following properties:
  M1. for any object $\mathcal{D}'$ of $\mu\nu\mathsf{Poly}_{\mathcal{D}}$, the unique functor $\mathcal{D}' \to \mathbb{1}$ is a morphism of $\mu\nu\mathsf{Poly}_{\mathcal{D}}$;
  M2. for any object $\mathcal{D}'$ of $\mu\nu\mathsf{Poly}_{\mathcal{D}}$, all the functors $\mathbb{1} \to \mathcal{D}'$ are morphisms of $\mu\nu\mathsf{Poly}_{\mathcal{D}}$;
  M3. the binary product $\times : \mathcal{D} \times \mathcal{D} \to \mathcal{D}$ is a morphism of $\mu\nu\mathsf{Poly}_{\mathcal{D}}$;
  M4. the binary coproduct $\sqcup : \mathcal{D} \times \mathcal{D} \to \mathcal{D}$ is a morphism of $\mu\nu\mathsf{Poly}_{\mathcal{D}}$;
  M5. for any pair of objects $(\mathcal{D}', \mathcal{D}'') \in \mu\nu\mathsf{Poly}_{\mathcal{D}} \times \mu\nu\mathsf{Poly}_{\mathcal{D}}$, the projections

$$\pi_1 : \mathcal{D}' \times \mathcal{D}'' \to \mathcal{D}', \qquad \pi_2 : \mathcal{D}' \times \mathcal{D}'' \to \mathcal{D}''$$

  are morphisms of $\mu\nu\mathsf{Poly}_{\mathcal{D}}$;
  M6. given objects $\mathcal{D}', \mathcal{D}'', \mathcal{D}'''$ of $\mu\nu\mathsf{Poly}_{\mathcal{D}}$, if $E : \mathcal{D}' \to \mathcal{D}''$ and $J : \mathcal{D}' \to \mathcal{D}'''$ are morphisms of $\mu\nu\mathsf{Poly}_{\mathcal{D}}$, then so is the induced functor $(E, J) : \mathcal{D}' \to \mathcal{D}'' \times \mathcal{D}'''$;
  M7. if $\mathcal{D}'$ is an object of $\mu\nu\mathsf{Poly}_{\mathcal{D}}$, $H : \mathcal{D}' \times \mathcal{D} \to \mathcal{D}$ is a morphism of $\mu\nu\mathsf{Poly}_{\mathcal{D}}$ and $\mu H : \mathcal{D}' \to \mathcal{D}$ exists, then μH is a morphism of $\mu\nu\mathsf{Poly}_{\mathcal{D}}$;
  M8. if $\mathcal{D}'$ is an object of $\mu\nu\mathsf{Poly}_{\mathcal{D}}$, $H : \mathcal{D}' \times \mathcal{D} \to \mathcal{D}$ is a morphism of $\mu\nu\mathsf{Poly}_{\mathcal{D}}$ and $\nu H : \mathcal{D}' \to \mathcal{D}$ exists, then νH is a morphism of $\mu\nu\mathsf{Poly}_{\mathcal{D}}$.

We say that $\mathcal{D}$ has *$\mu\nu$-polynomials* if $\mathcal{D}$ has finite coproducts and products and, for any endomorphism $(E : \mathcal{D} \to \mathcal{D})$ of $\mu\nu\mathsf{Poly}_{\mathcal{D}}$, 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.

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

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

Lemma 1.2. *Assume that $\mathcal{C}$ has $\mu\nu$-polynomials. In this case, if $\mathcal{D}$ is an object of $\mu\nu\mathsf{Poly}_{\mathcal{C}}$ and $H : \mathcal{D} \times \mathcal{C} \to \mathcal{C}$ is a morphism of $\mu\nu\mathsf{Poly}_{\mathcal{C}}$, then $\mu H : \mathcal{D} \to \mathcal{C}$ and $\nu H : \mathcal{D} \to \mathcal{C}$ exist (and, hence, they are morphisms of $\mu\nu\mathsf{Poly}_{\mathcal{C}}$).*

Proof: By Proposition D.1, it is enough to show that, for each $X \in \mathcal{D}$, μH^X and νH^X exist.

In fact, denoting by $X : \mathbb{1} \to \mathcal{D}$ the functor constantly equal to $X \in \mathcal{D}$, the functor H^X is the composition below.

$$\begin{tikzcd} \mathcal{C} \arrow[r, "{(1,\mathrm{id}_{\mathcal{C}})}"] \arrow[rrr, bend right, "H^X"'] & \mathbb{1} \times \mathcal{C} \arrow[r, "{(X\circ\pi_1,\mathrm{id}_{\mathcal{C}}\circ\pi_2)}"] & \mathcal{D} \times \mathcal{C} \arrow[r, "H"] & \mathcal{C} \end{tikzcd}$$

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

Definition 1.3. Let $\mathcal{D}$ be a category $\mu\nu$-polynomials. We say that a functor $G : \mathcal{D} \to \mathcal{C}$ *(strictly) preserves $\mu\nu$-polynomials* if it (strictly) preserves finite coproducts, finite products, initial algebras of $\mu\nu$-polynomials and terminal coalgebras of $\mu\nu$-polynomials (see Definitions E.3 and E.7).

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.

2. 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 $\mathsf{Op}^m_{n_1,\ldots,n_k}$ of primitive operations op, for all $k, m, n_1, \ldots, n_k \in \mathbb{N}$. These operations op will be interpreted as C^∞-smooth functions $(\mathbb{R}^{n_1} \times \cdots \times \mathbb{R}^{n_k}) \to \mathbb{R}^m$ and the reader can keep the following examples in mind:

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

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 $\alpha_1 : \text{type}, \ldots, \alpha_n : \text{type}$. The idea is that the type variables identifiers $\alpha_1, \ldots, \alpha_n$ can be used in the formation of τ. These kinding judgements 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 $x_1 : \tau_1, \ldots, x_n : \tau_n$ for variable identifiers x_i and types τ_i that are well-kinded in kinding context Δ. These typing judgements are defined according to the rules displayed in Fig. 3. As Fig. 5 displays, we consider the terms of our language up to the standard $\beta\eta$-theory. To present this equational theory, we define in Fig. 4, by induction, some syntactic sugar for the functorial action $\Delta, \Delta' \mid \Gamma, x : \tau[^{\sigma}/_{\alpha}] \vdash \tau[^{x \vdash t}/_{\alpha}] : \tau[^{\rho}/_{\alpha}]$ 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).

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

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

$$
\frac{((\alpha : \mathrm{type}) \in \Delta)}{\Delta \vdash \alpha : \mathrm{type}} \qquad
\frac{}{\Delta \vdash \mathbf{real}^n : \mathrm{type}} \qquad
\frac{}{\Delta \vdash \mathbf{1} : \mathrm{type}} \qquad
\frac{\Delta \vdash \tau : \mathrm{type} \quad \Delta \vdash \sigma : \mathrm{type}}{\Delta \vdash \tau {*} \sigma : \mathrm{type}}
$$

$$
\frac{\cdot \vdash \tau : \mathrm{type} \quad \cdot \vdash \sigma : \mathrm{type}}{\cdot \vdash \tau \to \sigma : \mathrm{type}} \qquad
\frac{\{\Delta \vdash \tau_i : \mathrm{type} \quad \ell_i \ \mathrm{label}\}_{1 \le i \le n}}{\Delta \vdash \{\ell_1 \tau_1 \mid \cdots \mid \ell_n \tau_n\} : \mathrm{type}}
$$

$$
\frac{\Delta, \alpha : \kappa \vdash \tau : \mathrm{type}}{\Delta \vdash \mu\alpha.\tau : \mathrm{type}} \qquad
\frac{\Delta, \alpha : \kappa \vdash \tau : \mathrm{type}}{\Delta \vdash \nu\alpha.\tau : \mathrm{type}}
$$

FIGURE 2. Kinding rules for the AD source language. Note that we only consider the formation of function types of non-parameterized types (shaded in grey).

This standard language is equivalent to the freely generated bi-Cartesian closed category **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

$$\frac{((x:\tau)\in\Gamma)}{\Delta\mid\Gamma\vdash x:\tau}\quad \frac{\Delta\mid\Gamma\vdash t:\tau\quad \Delta\mid\Gamma,x:\tau\vdash s:\sigma}{\Delta\mid\Gamma\vdash \mathbf{let}\,x=\,t\,\mathbf{in}\,s:\sigma}\quad \frac{\{\Delta\mid\Gamma\vdash t_i:\mathbf{real}^{n_i}\}_{i=1}^k\quad (\mathrm{op}\in\mathsf{Op}^m_{n_1,\ldots,n_k})}{\Delta\mid\Gamma\vdash \mathrm{op}(t_1,\ldots,t_k):\mathbf{real}^m}$$

$$\frac{}{\Delta\mid\Gamma\vdash\langle\rangle:\mathbf{1}}\quad \frac{\Delta\mid\Gamma\vdash t:\tau\quad \Delta\mid\Gamma\vdash s:\sigma}{\Delta\mid\Gamma\vdash\langle t,s\rangle:\tau\boldsymbol{*}\sigma}\quad \frac{\Delta\mid\Gamma\vdash t:\tau\boldsymbol{*}\sigma}{\Delta\mid\Gamma\vdash\mathbf{fst}\,t:\tau}\quad \frac{\Delta\mid\Gamma\vdash t:\tau\boldsymbol{*}\sigma}{\Delta\mid\Gamma\vdash\mathbf{snd}\,t:\sigma}$$

$$\frac{\Delta\mid\Gamma,x:\tau\vdash t:\sigma}{\Delta\mid\Gamma\vdash\lambda x.t:\tau\to\sigma}\quad \frac{\Delta\mid\Gamma\vdash t:\sigma\to\tau\qquad \Delta\mid\Gamma\vdash s:\sigma}{\Delta\mid\Gamma\vdash t\,s:\tau}$$

$$\frac{\Delta\mid\Gamma\vdash t:\tau_i}{\Delta\mid\Gamma\vdash\ell_i t:\{\ell_1\tau_1\mid\cdots\mid\ell_n\tau_n\}}\quad \frac{\Delta\mid\Gamma\vdash t:\{\ell_1\tau_1\mid\cdots\mid\ell_n\tau_n\}\quad \{\Delta\mid\Gamma,x_i:\tau_i\vdash s_i:\rho\}_{1\le i\le n}}{\Delta\mid\Gamma\vdash\mathbf{case}\,t\,\mathbf{of}\,\{\ell_1x_1\to s_1\mid\cdots\mid\ell_nx_n\to s_n\}:\rho}$$

$$\frac{\Delta\mid\Gamma\vdash t:\tau[^{\mu\alpha.\tau}/_{\alpha}]}{\Delta\mid\Gamma\vdash\mathbf{roll}\,t:\mu\alpha.\tau}\quad \frac{\Delta\mid\Gamma\vdash t:\mu\alpha.\tau\quad \Delta\mid x:\tau[^{\sigma}/_{\alpha}]\vdash s:\sigma}{\Delta\mid\Gamma\vdash\mathbf{fold}\,t\,\mathbf{with}\,x\to s:\sigma}$$

$$\frac{\Delta\mid\Gamma\vdash t:\sigma\quad \Delta\mid x:\sigma\vdash s:\tau[^{\sigma}/_{\alpha}]}{\Delta\mid\Gamma\vdash\mathbf{gen\,from}\,t\,\mathbf{with}\,x\to s:\nu\alpha.\tau}\quad \frac{\Delta\mid\Gamma\vdash t:\nu\alpha.\tau}{\Delta\mid\Gamma\vdash\mathbf{unroll}\,t:\tau[^{\nu\alpha.\tau}/_{\alpha}]}$$

FIGURE 3. Typing rules for the AD source language.

$$\begin{aligned}
\alpha[^{x\vdash t}/_{\alpha}] &= t\\
\beta[^{x\vdash t}/_{\alpha}] &= x \qquad \text{if } \alpha\neq\beta\\
\mathbf{real}^n[^{x\vdash t}/_{\alpha}] &= x\\
\mathbf{1}[^{x\vdash t}/_{\alpha}] &= x\\
(\tau\boldsymbol{*}\sigma)[^{x\vdash t}/_{\alpha}] &= \langle\tau[^{x\vdash t}/_{\alpha}][^{\mathbf{fst}\,x}/_{x}],\sigma[^{x\vdash t}/_{\alpha}][^{\mathbf{snd}\,x}/_{x}]\rangle\\
\{\ell_1\tau_1\mid\cdots\mid\ell_n\tau_n\}[^{x\vdash t}/_{\alpha}] &= \mathbf{case}\,x\,\mathbf{of}\,\{\ell_1x\to\ell_1\tau_1[^{x\vdash\ell_1 t}/_{\alpha}]\mid\cdots\mid\ell_nx\to\ell_n\tau_n[^{x\vdash\ell_n t}/_{\alpha}]\}\\
(\mu\alpha.\tau)[^{x\vdash t}/_{\alpha}] &= x\\
(\mu\beta.\tau)[^{x\vdash t}/_{\alpha}] &= \mathbf{fold}\,x\,\mathbf{with}\,x\to\mathbf{roll}\,\tau[^{x\vdash t}/_{\alpha}] \qquad \text{if } \alpha\neq\beta\\
(\nu\alpha.\tau)[^{x\vdash t}/_{\alpha}] &= x\\
(\nu\beta.\tau)[^{x\vdash t}/_{\alpha}] &= \mathbf{gen\,from}\,x\,\mathbf{with}\,x\to\tau[^{x\vdash t}/_{\alpha}][^{\mathbf{unroll}\,x}/_{x}] \qquad \text{if } \alpha\neq\beta
\end{aligned}$$

FIGURE 4. Functorial action $\Delta,\Delta'\mid\Gamma,x:\tau[^{\sigma}/_{\alpha}]\vdash\tau[^{x\vdash t}/_{\alpha}]:\tau[^{\rho}/_{\alpha}]$ in argument α of parameterized types $\Delta,\alpha:\text{type}\vdash\tau:\text{type}$ on terms $\Delta'\mid\Gamma,x:\sigma\vdash t:\rho$ of the source language.

supports tuple types, function types, sum types, inductive and coinductive types and primitive types $\mathsf{Ty}=\{\mathbf{real}^n\mid n\in\mathbb{N}\}$ and primitive operations $\mathsf{Op}(\mathbf{real}^{n_1},\ldots,\mathbf{real}^{n_k};\mathbf{real}^m)=\mathsf{Op}^m_{n_1,\ldots,n_k}$ (in the sense of §1). **Syn** effectively represents programs as (categorical) combinators, also known as "point-free style" in the functional programming community. Concretely, **Syn** has types as objects, homsets $\mathbf{Syn}(\tau,\sigma)$ consist of $(\alpha)\beta\eta$-equivalence classes of terms

$$\mathbf{let}\,x =\ t\,\mathbf{in}\,s = s[{}^{t}/_{x}] \qquad t = \langle\rangle \qquad \mathbf{fst}\,\langle t, s\rangle = t \qquad \mathbf{snd}\,\langle t, s\rangle = s \qquad t = \langle \mathbf{fst}\,t, \mathbf{snd}\,t\rangle$$

$$(\lambda x.t)\,s = t[{}^{s}/_{x}] \qquad t \overset{\#x}{=} \lambda x.t\,x \qquad \mathbf{case}\,\ell_i t\,\mathbf{of}\,\{\ell_1 x_1 \to s_1 \mid \cdots \mid \ell_n x_n \to s_n\} = s_i[{}^{t}/_{x_i}]$$

$$s[{}^{t}/_{y}] \overset{\#x_1,\ldots,x_n}{=} \mathbf{case}\,t\,\mathbf{of}\,\{\ell_1\,x_1 \to s[{}^{\ell_1\,x_1}/_{y}] \mid \cdots \mid \ell_\mathsf{n}\,x_n \to s[{}^{\ell_\mathsf{n}\,x_n}/_{y}]\}$$

$$\mathbf{fold}\,\mathbf{roll}\,t\,\mathbf{with}\,x \to s \overset{\#y}{=} s[{}^{\tau[{}^{y \vdash \mathbf{fold}\,y\,\mathbf{with}\,x \to s}/_{\alpha}][{}^{t}/_{y}]}/_{x}]$$

$$r[{}^{\mathbf{roll}\,x}/_{x}] = s[{}^{\tau[{}^{x \vdash r}/_{\alpha}]}/_{z}] \Rightarrow r[{}^{t}/_{x}] = \mathbf{fold}\,t\,\mathbf{with}\,z \to s$$

$$\mathbf{unroll}\,(\mathbf{gen}\,\mathbf{from}\,t\,\mathbf{with}\,x \to s) \overset{\#y}{=} \tau[{}^{y \vdash \mathbf{gen}\,\mathbf{from}\,y\,\mathbf{with}\,x \to s}/_{\alpha}][{}^{s}/_{y}, {}^{t}/_{x}]$$

$$\mathbf{unroll}\,r = \tau[{}^{x \vdash r}/_{\alpha}][{}^{s}/_{x}] \Rightarrow r[{}^{t}/_{x}] = \mathbf{gen}\,\mathbf{from}\,t\,\mathbf{with}\,x \to s$$

FIGURE 5. We consider the standard $\beta\eta$-laws above for our language. We write $\overset{\#x_1,\ldots,x_n}{=}$ to indicate that the variables $x_1, \ldots, x_n$ 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.

$\cdot \mid x : \tau \vdash t : \sigma$, 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 $\cdot \mid x : \tau \vdash \mathbf{let}\,y =\ t\,\mathbf{in}\,s : \rho$.

Corollary 2.1 (Universal property of **Syn**)**.** *Given any bi-Cartesian closed category with $\mu\nu$-polynomials $\mathcal{C}$, any consistent assignment of*

$$F(\mathbf{real}^n) \in \mathrm{ob}\,\mathcal{C}$$

and $F(\mathrm{op}) \in \mathcal{C}(F(\mathbf{real}^{n_1}) \times \cdots \times F(\mathbf{real}^{n_k}), F(\mathbf{real}^{m}))$ *for* $\mathrm{op} \in \mathsf{Op}^m_{n_1,\ldots,n_k}$ *extends to a unique $\mu\nu$-polynomial preserving bi-Cartesian closed functor*

$$F : \mathbf{Syn} \to \mathcal{C}.$$

3. Modelling expressive functional languages in Grothendieck constructions

In this section, we present a novel construction of categorical models (in the sense of §1) $\Sigma_{\mathcal{C}}\mathcal{L}$ and $\Sigma_{\mathcal{C}}\mathcal{L}^{op}$ of expressive functional languages (like our AD source language of §2) in Σ-types of suitable indexed categories

$$\mathcal{L} : \mathcal{C}^{op} \to \mathbf{Cat}.$$

In particular, the problem we solve in this section is to identify suitable sufficient conditions to put on an indexed category $\mathcal{L} : \mathcal{C}^{op} \to \mathbf{Cat}$, whose base category we think of as the semantics of a Cartesian type theory and whose fibre categories we think of as the semantics of a dependent linear type theory, such that $\Sigma_{\mathcal{C}}\mathcal{L}$ and $\Sigma_{\mathcal{C}}\mathcal{L}^{op}$ 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 satifies our sufficient conditions for $\Sigma_{\mathcal{C}}\mathcal{L}$ and $\Sigma_{\mathcal{C}}\mathcal{L}^{op}$ 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 $\mathbf{LSyn} : \mathbf{CSyn}^{op} \to \mathbf{Cat}$ of a suitable target language for AD translations (§4); the universal property of the source language $\mathbf{Syn}$ then yields unique structure preserving functors $\overrightarrow{\mathcal{D}} : \mathbf{Syn} \to \Sigma_{\mathbf{CSyn}}\mathbf{LSyn}$ and $\overleftarrow{\mathcal{D}} : \mathbf{Syn} \to \Sigma_{\mathbf{CSyn}}\mathbf{LSyn}^{op}$ implementing forward and reverse mode AD;
- the case where $\mathcal{L}$ is the indexed category of families of real vector spaces $\mathbf{FVect} : \mathbf{Set}^{op} \to \mathbf{Cat}$ (§6); this gives a concrete denotational semantics to the target language, which we use in the correctness proof of AD.

3.1. Basics: the categories $\Sigma_{\mathcal{C}}\mathcal{L}$ and $\Sigma_{\mathcal{C}}\mathcal{L}^{\mathrm{op}}$. Recall that for any strictly indexed category, i.e. a (strict) functor $\mathcal{L} : \mathcal{C}^{\mathrm{op}} \to \mathbf{Cat}$, we can consider its total category (or Grothendieck construction) $\Sigma_{\mathcal{C}}\mathcal{L}$, which is a fibred category over $\mathcal{C}$ (see [20, 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 $\mathcal{L} : \mathcal{C}^{\mathrm{op}} \to \mathbf{Cat}$, we can consider its fibrewise dual category $\mathcal{L}^{\mathrm{op}} : \mathcal{C}^{\mathrm{op}} \to \mathbf{Cat}$, which is defined as the composition $\mathcal{C}^{\mathrm{op}} \xrightarrow{\mathcal{L}} \mathbf{Cat} \xrightarrow{\mathrm{op}} \mathbf{Cat}$. Thus, we can apply the same construction to $\mathcal{L}^{\mathrm{op}}$ to obtain a category $\Sigma_{\mathcal{C}}\mathcal{L}^{\mathrm{op}}$.

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

- objects are pairs (A_1, A_2) of an object A_1 of $\mathcal{C}$ and an object A_2 of $\mathcal{L}(A_2)$;
- morphisms $(A_1, A_2) \to (B_1, B_2)$ are pairs (f_1, f_2) with $f_1 : A_1 \to B_1$ in $\mathcal{C}$ and $f_2 : A_2 \to \mathcal{L}(f_1)(B_2)$ in $\mathcal{L}(A_1)$;
- identities $\mathrm{id}_{(A_1,A_2)}$ are $(\mathrm{id}_{A_1}, \mathrm{id}_{A_2})$;
- composition of $(A_1, A_2) \xrightarrow{(f_1,f_2)} (B_1, B_2)$ and $(B_1, B_2) \xrightarrow{(g_1,g_2)} (C_1, C_2)$ is given by
$(f_1; g_1, f_2; \mathcal{L}(f_1)(g_2))$.

Concretely, $\Sigma_{\mathcal{C}}\mathcal{L}^{\mathrm{op}}$ is the following category:

- objects are pairs (A_1, A_2) of an object A_1 of $\mathcal{C}$ and an object A_2 of $\mathcal{L}(A_1)$;
- morphisms $(A_1, A_2) \to (B_1, B_2)$ are pairs (f_1, f_2) with $f_1 : A_1 \to B_1$ in $\mathcal{C}$ and $f_2 : \mathcal{L}(f_1)(B_2) \to A_2$ in $\mathcal{L}(A_1)$;
- identities $\mathrm{id}_{(A_1,A_2)}$ are $(\mathrm{id}_{A_1}, \mathrm{id}_{A_2})$;
- composition of $(A_1, A_2) \xrightarrow{(f_1,f_2)} (B_1, B_2)$ and $(B_1, B_2) \xrightarrow{(g_1,g_2)} (C_1, C_2)$ is given by
$(f_1; g_1, \mathcal{L}(f_1)(g_2); f_2)$.

3.2. Product structure. We say that a strictly indexed category $\mathcal{L}$ has *strictly indexed finite (co)products* if

- each fibre $\mathcal{L}(C)$ has chosen finite (co)products $(\mathbb{1}, \times)$;
- change of base strictly preserves these (co)products in the sense that $\mathcal{L}(f)(\mathbb{1}) = \mathbb{1}$ and $\mathcal{L}(f)(A \times B) = \mathcal{L}(f)(A) \times \mathcal{L}(f)(B)$ for all morphisms f in $\mathcal{C}$.

We recall below that $\Sigma_{\mathcal{C}}\mathcal{L}$ has finite products if $\mathcal{C}$ has finite products and $\mathcal{L}$ has finite indexed products.

Proposition 3.1. *Assuming that $\mathcal{C}$ has finite products $(\mathbb{1}, \times)$ and $\mathcal{L}$ has indexed finite products $(\mathbb{1}, \times)$, $\Sigma_{\mathcal{C}}\mathcal{L}$ has (fibred) terminal object $\mathbb{1} = (\mathbb{1}, \mathbb{1})$ and (fibred) 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

$$
\begin{aligned}
&\Sigma_{\mathcal{C}}\mathcal{L}((X,x),(\mathbb{1},\mathbb{1})) \\
&= \Sigma_{f_1 \in \mathcal{C}(X,\mathbb{1})}\mathcal{L}(X)(x, \mathcal{L}(f_1)(\mathbb{1})) \\
&\cong \Sigma_{f_1 \in \mathcal{C}(X,\mathbb{1})}\mathcal{L}(X)(x, \mathbb{1}) && \{\ \text{indexed } \mathbb{1}\ \} \\
&\cong \mathbb{1} \times \mathbb{1} && \{\ \mathbb{1} \text{ terminal in } \mathcal{C} \text{ and } \mathcal{L}(X)\ \} \\
&\cong \mathbb{1}
\end{aligned}
$$

$$
\begin{aligned}
&\Sigma_{\mathcal{C}}\mathcal{L}\left((X,x),(W \times Z, \mathcal{L}(\pi_1)(w) \times \mathcal{L}(\pi_2)(z))\right) \\
&= \Sigma_{(f_1,g_1) \in \mathcal{C}(X, W \times Y)}\mathcal{L}(X)(x, \mathcal{L}((f_1,g_1))(\mathcal{L}(\pi_1)(w) \times \mathcal{L}(\pi_2)(z))) \\
&\cong \Sigma_{(f_1,g_1) \in \mathcal{C}(X, W \times Z)}\mathcal{L}(X)(x, \mathcal{L}((f_1,g_1))(\mathcal{L}(\pi_1)(w)) \times \mathcal{L}((f_1,g_1))(\mathcal{L}(\pi_2)(z))) && \{\ \text{indexed } \times\ \} \\
&= \Sigma_{(f_1,g_1) \in \mathcal{C}(X, W \times Z)}\mathcal{L}(X)(x, \mathcal{L}(f_1)(w) \times \mathcal{L}(g_1)(z)) && \{\ \text{functoriality } \mathcal{L}\ \} \\
&\cong \Sigma_{(f_1,g_1) \in \mathcal{C}(X, W \times Z)}\mathcal{L}(X)(x, \mathcal{L}(f_1)(w)) \times \mathcal{L}(X)(x, \mathcal{L}(g_1)(z)) && \{\ \times \text{ product in } \mathcal{L}(A_1)\ \} \\
&\cong \Sigma_{f_1 \in \mathcal{C}(X,W)}\Sigma_{g_1 \in \mathcal{C}(X,Z)}\mathcal{L}(X)(x, \mathcal{L}(f_1)(w)) \times \mathcal{L}(X)(x, \mathcal{L}(g_1)(z)) && \{\ \times \text{ product in } \mathcal{C}\ \} \\
&\cong \left(\Sigma_{f_1 \in \mathcal{C}(X,W)}\mathcal{L}(X)(x, \mathcal{L}(f_1)(w))\right) \times \left(\Sigma_{g_1 \in \mathcal{C}(X,Z)}\mathcal{L}(X)(x, \mathcal{L}(g_1)(z))\right) && \{\ \text{Beck-Chevalley for } \Sigma \text{ in } \mathbf{Set}\ \} \\
&= \Sigma_{\mathcal{C}}\mathcal{L}((X,x),(W,w)) \times \Sigma_{\mathcal{C}}\mathcal{L}((X,x),(Z,z)).
\end{aligned}
$$

■

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

Codually, we have:

Proposition 3.2. *Assuming that $\mathcal{C}$ has finite products $(\mathbb{1}, \times)$ and $\mathcal{L}$ has finite indexed coproducts $(\mathbb{0}, \sqcup)$, we have that $\Sigma_{\mathcal{C}}\mathcal{L}^{\mathrm{op}}$ has (fibred) terminal object $\mathbb{1} = (\mathbb{1}, \mathbb{0})$ and (fibred) binary product $(A_1, A_2) \times (B_1, B_2) = (A_1 \times B_1, \mathcal{L}(\pi_1)(A_2) \sqcup \mathcal{L}(\pi_2)(B_2))$.*

That is, in our terminology, $\mathcal{L} : \mathcal{C}^{op} \to \mathbf{Cat}$ 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 these the results in this section in the situation where $\mathcal{L}$ has indexed finite biproducts (products that are simultaneously coproducts), in which case the finite product structures of $\Sigma_{\mathcal{C}}\mathcal{L}$ and $\Sigma_{\mathcal{C}}\mathcal{L}^{op}$ coincide.

3.3. 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 §1. 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.

That is, we say that $\mathcal{L} : \mathcal{C}^{op} \to \mathbf{Cat}$ is a Σ-*bimodel of primitive types* Ty *and operations* Op if

- we have, for all $T \in \mathsf{Ty}$, a choice of objects $C_T \in \mathrm{ob}\,\mathcal{C}$ and $L_T, L'_T \in \mathrm{ob}\,\mathcal{L}(C_T)$;
- we have, for all $\mathrm{op} \in \mathsf{Op}(T_1, \ldots, T_n; S)$, a choice of morphisms

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

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

3.4. Closed structure. In this section, 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 [38, Chapter 5] and [3].

We briefly recall some of the usual vocabulary here. Given an indexed category $\mathcal{D} : \mathcal{C}^{op} \to \mathbf{Cat}$, we say

- it satisfies the *comprehension axiom* if $\mathcal{C}$ has a chosen terminal object $\cdot$, $\mathcal{D}$ has strictly indexed terminal objects $\mathbb{1}$ (i.e. chosen terminal objects $\mathbb{1} \in \mathcal{D}(C)$, such that $\mathcal{C}(f)(\mathbb{1}) = \mathbb{1}$ for all $f : C' \to C$ in $\mathcal{C}$) and the functors
$$(\mathcal{C}/A)^{op} \to \mathbf{Set}$$
$$(C \xrightarrow{f} A) \mapsto \mathcal{D}(C)(\mathbb{1}, \mathcal{D}(f)(B))$$
are representable by a chosen object $\mathbf{p}_{A,B} : A.B \to A$ of $\mathcal{C}/A$:
$$\begin{aligned}\mathcal{D}(C)(\mathbb{1}, \mathcal{C}'(f)(B)) &\cong \mathcal{C}/A(f, \mathbf{p}_{A,B})\\ b &\mapsto (f, b);\end{aligned}$$
we write $\mathbf{v}_{A,B}$ for the unique element of $\mathcal{D}(A.B)(\mathbb{1}, \mathcal{D}(\mathbf{p}_{A,B})(B))$ such that $(\mathbf{p}_{A,B}, \mathbf{v}_{A,B}) = \mathrm{id}_{\mathbf{p}_{A,B}}$ (the universal element of the representation); further, given $f : A' \to A$, we write $\mathbf{q}_{f,B}$ for the unique morphism $(\mathbf{p}_{A',\mathcal{D}(f)(B)}; f, \mathbf{v}_{A',\mathcal{D}(f)(B)})$ making the square below a pullback; we henceforth call such squares $\mathbf{p}$-squares;
$$\begin{tikzcd} A'.\mathcal{D}(f)(B) \arrow[r, "\mathbf{q}_{f,B}"] \arrow[d, "\mathbf{p}_{A',\mathcal{D}(f)(B)}"'] & A.B \arrow[d, "\mathbf{p}_{A,B}"] \\ A' \arrow[r, "f"'] & A \end{tikzcd}$$
- it *supports (weak)* Σ*-types* if we have left adjoint functors $\Sigma_B \dashv \mathcal{D}(\mathbf{p}_{A,B}) : \mathcal{D}(A.B) \leftrightarrows \mathcal{D}(A)$ satisfying the left Beck-Chevalley condition for $\mathbf{p}$-squares in the sense that the canonical maps $\mathcal{D}(\mathbf{q}_{f,B}); \Sigma_{\mathcal{D}(f)(B)} \to \Sigma_B; \mathcal{D}(f)$ are isomorphisms;
- it *supports* Π*-types* if $\mathcal{D}^{op}$ supports (weak) Σ-types; explicitly, that is the case iff we have right adjoint functors $\mathcal{D}(\mathbf{p}_{A,B}) \dashv \Pi_B : \mathcal{D}(A) \leftrightarrows \mathcal{D}(A.B)$ satisfying the right Beck-Chevalley condition for $\mathbf{p}$-squares in the sense that the canonical maps $\Pi_B; \mathcal{D}(f) \to \mathcal{D}(\mathbf{q}_{f,B}); \Pi_{\mathcal{D}(f)(B)}$ are isomorphisms.

In case $\mathcal{D}$ satisfies the comprehension axiom, we further say that

- it satisfies *democratic comprehension* if the comprehension functor
$$\begin{aligned}\mathcal{D}(A)(B', B) &\xrightarrow{\mathbf{p}_{A,-}} \mathcal{C}/A(\mathbf{p}_{A,B'}, \mathbf{p}_{A,B})\\ d &\mapsto (\mathbf{p}_{A,B'}, \mathbf{v}_{A,B'}; \mathcal{D}(\mathbf{p}_{A,B'})(d))\end{aligned}$$

defines an equivalence of categories $\mathcal{D}(\cdot) \cong \mathcal{C}/\cdot \cong \mathcal{C}$;
- 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 weak Σ-types) if dependent projections compose: for all B, C, D, we have for some objects $\Sigma_C D$ of $\mathcal{D}(B)$ such that $\mathbf{p}_{B.C,D}; \mathbf{p}_{B,C} = \mathbf{p}_{B,\Sigma_C D}$.

Definition 3.3 (Σ-bimodel of function types)**.** We call a strictly indexed category $\mathcal{L} : \mathcal{C}^{op} \to \mathbf{Cat}$ a Σ*-bimodel of function types* if it is a biadditive model of the dependently typed enriched effect calculus in the sense that it comes equipped with
- a model of Cartesian dependent type theory in the sense of a strictly indexed category $\mathcal{C}' : \mathcal{C}^{op} \to \mathbf{Cat}$ that satisfies full, faithful, democratic comprehension with Π-types and strong Σ-types;
- strictly indexed finite biproducts $(\mathbb{1}, \times)$ and Σ- and Π-types in $\mathcal{L}$;
- a strictly indexed functor $\multimap: \mathcal{L}^{\mathrm{op}} \times \mathcal{L} \to \mathcal{C}'$ and a natural isomorphism
$$\mathcal{L}(A)(B, C) \cong \mathcal{C}'(A)(\mathbb{1}, B \multimap C).$$

We can immediately note that a 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 in the sense that it also gives us Cartesian closure of the corresponding Grothendieck constructions. We generalize the proofs from [39] here, to make sure that they also apply to the case where $\mathcal{L}$ is a general strictly indexed category rather than a locally indexed one.

In the following, we will slightly abuse notation to aid legibility:
- we will sometimes conflate $B \in \mathrm{ob}\, \mathcal{C}'(\cdot)$ and $\cdot.B \in \mathrm{ob}\, \mathcal{C}$ as well as $f \in \mathcal{C}'(C)(\mathbb{1}, \mathcal{C}'(!_C)(B))$ and $(!_C, f) \in \mathcal{C}(C, \cdot.B)$); this is justified because of the democracy of the comprehension;
- we will sometimes simply write C for $\mathcal{D}(\mathbf{p}_{A,B})(C)$ where the weakening map $\mathcal{D}(\mathbf{p}_{A,B})$ is clear from context.

Given $A_1, B_1 \in \mathcal{C}$ we will write ev1 for the obvious morphism
$$\mathrm{ev1} : \Pi_{A_1}\Sigma_{B_1} D.A_1 \to B_1.$$

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

Theorem 3.1. *For a Σ-bimodel $\mathcal{L}$ of function types, $\Sigma_{\mathcal{C}}\mathcal{L}$ has exponential*

$$(A_1, A_2) \Rightarrow (B_1, B_2) = (\Pi_{A_1}\Sigma_{B_1}\mathcal{L}(\pi_1)(A_2) \multimap \mathcal{L}(\pi_2)(B_2), \Pi_{A_1}\mathcal{L}(\mathrm{ev}1)(B_2)).$$

Proof: We have (natural) bijections

$$\begin{aligned}
&\Sigma_{\mathcal{C}}\mathcal{L}((A_1, A_2) \times (B_1, B_2), (C_1, C_2)) = \\
&= \Sigma_{\mathcal{C}}\mathcal{L}((A_1 \times B_1, \mathcal{L}(\pi_1)(A_2) \times \mathcal{L}(\pi_2)(B_2)), (C_1, C_2)) \\
&= \Sigma_{f_1 \in \mathcal{C}(A_1 \times B_1, C_1)}\mathcal{L}(A_1 \times B_1)(\mathcal{L}(\pi_1)(A_2) \times \mathcal{L}(\pi_2)(B_2), \mathcal{L}(f_1)(C_2)) \\
&\cong \Sigma_{f_1 \in \mathcal{C}(A_1 \times B_1, C_1)}\mathcal{L}(A_1 \times B_1)(\mathcal{L}(\pi_1)(A_2), \mathcal{L}(f_1)(C_2)) \times \mathcal{L}(A_1 \times B_1)(\mathcal{L}(\pi_2)(B_2), \mathcal{L}(f_1)(C_2)) && \{\ \times \text{ coproduct in } \mathcal{L}(A_1 \times B_1)\ \} \\
&\cong \Sigma_{f_1 \in \mathcal{C}(A_1 \times B_1, C_1)}\mathcal{L}(A_1)(A_2, \Pi_{B_1}\mathcal{L}(f_1)(C_2)) \times \mathcal{L}(A_1 \times B_1)(\mathcal{L}(\pi_2)(B_2), \mathcal{L}(f_1)(C_2)) && \{\ \Pi\text{-types in } \mathcal{L}\ \} \\
&\cong \Sigma_{f_1 \in \mathcal{C}(A_1 \times B_1, C_1)}\mathcal{L}(A_1)(A_2, \Pi_{B_1}\mathcal{L}(f_1)(C_2)) \times \mathcal{C}'(A_1 \times B_1)(\mathbb{1}, \mathcal{L}(\pi_2)(B_2) \multimap \mathcal{L}(f_1)(C_2)) && \{\ \multimap\text{-types in } \mathcal{C}'\ \} \\
&\cong \Sigma_{(f_1,g_1) \in \Sigma_{f_1 \in \mathcal{C}(A_1 \times B_1, C_1)}\mathcal{C}'(A_1 \times B_1)(\mathbb{1}, \mathcal{L}(\pi_2)(B_2) \multimap \mathcal{L}(f_1)(C_2))}\mathcal{L}(A_1)(A_2, \Pi_{B_1}\mathcal{L}(f_1)(C_2)) && \{\ \Sigma\text{-types in } \mathbf{Set}\ \} \\
&\cong \Sigma_{(f_1,g_1) \in \Sigma_{f_1 \in \mathcal{C}'(A_1 \times B_1)(\mathbb{1}, C_1)}\mathcal{C}'(A_1 \times B_1)(\mathbb{1}, \mathcal{L}(\pi_2)(B_2) \multimap \mathcal{L}(f_1)(C_2))}\mathcal{L}(A_1)(A_2, \Pi_{B_1}\mathcal{L}(f_1)(C_2)) && \{\ \text{comprehension}\ \} \\
&\cong \Sigma_{(f_1,g_1) \in \mathcal{C}'(A_1 \times B_1)(\mathbb{1}, \Sigma_{C_1}\mathcal{L}(\pi_1;\pi_2)(B_2) \multimap \mathcal{L}(\pi_2)(C_2))}\mathcal{L}(A_1)(A_2, \Pi_{B_1}\mathcal{L}(f_1)(C_2)) && \{\ \text{strong } \Sigma\text{-types in } \mathcal{C}'\ \} \\
&\cong \Sigma_{(f_1,g_1) \in \mathcal{C}'(A_1 \times B_1)(\mathbb{1}, \Sigma_{C_1}\mathcal{L}(\pi_1;\pi_2)(B_2) \multimap \mathcal{L}(\pi_2)(C_2))}\mathcal{L}(A_1)(A_2, \mathcal{L}((f_1, g_1))(\Pi_{B_1}\mathcal{L}(\mathrm{ev}1)(C_2))) && \{\ \text{Beck-Chevalley for } \Pi\text{-types}\ \} \\
&\cong \Sigma_{h_1 \in \mathcal{C}'(A_1 \times B_1)(\mathbb{1}, \Sigma_{C_1}\mathcal{L}(\pi_1;\pi_2)(B_2) \multimap \mathcal{L}(\pi_2)(C_2))}\mathcal{L}(A_1)(A_2, \mathcal{L}(h_1)(\Pi_{B_1}\mathcal{L}(\mathrm{ev}1)(C_2))) && \{\ \text{strong } \Sigma\text{-types in } \mathcal{C}'\ \} \\
&\cong \Sigma_{h_1 \in \mathcal{C}'(A_1 \times B_1)(\mathcal{L}(\pi_1)(\mathbb{1}), \Sigma_{C_1}\mathcal{L}(\pi_1;\pi_2)(B_2) \multimap \mathcal{L}(\pi_2)(C_2))}\mathcal{L}(A_1)(A_2, \mathcal{L}(h_1)(\Pi_{B_1}\mathcal{L}(\mathrm{ev}1)(C_2))) && \{\ \text{indexed } \mathbb{1} \text{ in } \mathcal{C}'\ \} \\
&\cong \Sigma_{h_1 \in \mathcal{C}'(A_1)(\mathbb{1}, \Pi_{B_1}\Sigma_{C_1}\mathcal{L}(\pi_1;\pi_2)(B_2) \multimap \mathcal{L}(\pi_2)(C_2))}\mathcal{L}(A_1)(A_2, \mathcal{L}(h_1)(\Pi_{B_1}\mathcal{L}(\mathrm{ev}1)(C_2))) && \{\ \Pi\text{-types in } \mathcal{C}'\ \} \\
&\cong \Sigma_{h_1 \in \mathcal{C}(A_1, \Pi_{B_1}\Sigma_{C_1}\mathcal{L}(\pi_1)(B_2) \multimap \mathcal{L}(\pi_2)(C_2))}\mathcal{L}(A_1)(A_2, \mathcal{L}(h_1)(\Pi_{B_1}\mathcal{L}(\mathrm{ev}1)(C_2))) && \{\ \text{comprehension}\ \} \\
&= \Sigma_{\mathcal{C}}\mathcal{L}((A_1, A_m2), (\Pi_{B_1}\Sigma_{C_1}\mathcal{L}(\pi_1)(B_2) \multimap \mathcal{L}(\pi_2)(C_2), \Pi_{B_1}\mathcal{L}(\mathrm{ev}1)(C_2))) \\
&= \Sigma_{\mathcal{C}}\mathcal{L}((A_1, A_2), (B_1, B_2) \Rightarrow (C_1, C_2)).
\end{aligned}$$

■

Codually, we have:

Theorem 3.2. *For a Σ-bimodel $\mathcal{L}$ of function types, $\Sigma_{\mathcal{C}}\mathcal{L}^{\mathrm{op}}$ has exponential* $(A_1, A_2) \Rightarrow (B_1, B_2) = (\Pi_{A_1}\Sigma_{B_1}\mathcal{L}(\pi_2)(B_2) \multimap \mathcal{L}(\pi_1)(A_2), \Sigma_{A_1}\mathcal{L}(\mathrm{ev}1)(B_2))$.

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

3.5. Coproduct structure. We introduce another special property that fits our context well. We call this property *extensivity* because it generalizes the concept of *extensive categories*.

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

$$W \xrightarrow{\iota_1 = \iota_W} W \sqcup X \xleftarrow{\iota_2 = \iota_X} X \tag{3.1}$$

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

Definition 3.4 (Extensive indexed categories)**.** We call an indexed category $\mathcal{L} : \mathcal{C}^{\mathrm{op}} \to \mathbf{Cat}$ *extensive* if, for any $(W, X) \in \mathcal{C} \times \mathcal{C}$, the unique functor

$$\mathcal{L}(W \sqcup X) \xrightarrow{(\mathcal{L}(\iota_W),\mathcal{L}(\iota_X))} \mathcal{L}(W) \times \mathcal{L}(X) \tag{3.2}$$

induced by the functors

$$\mathcal{L}(W) \xleftarrow{\mathcal{L}(\iota_W)} \mathcal{L}(W \sqcup X) \xrightarrow{\mathcal{L}(\iota_X)} \mathcal{L}(X) \tag{3.3}$$

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

$$\mathcal{S}^{(W,X)} : \mathcal{L}(W) \times \mathcal{L}(X) \to \mathcal{L}(W \sqcup X) \tag{3.4}$$

an inverse equivalence of $(\mathcal{L}(\iota_W), \mathcal{L}(\iota_X))$.

Since the products of $\mathcal{C}^{\mathrm{op}}$ are the coproducts of $\mathcal{C}$, the extensive condition described above is equivalent to say that the (pseudo)functor $\mathcal{L} : \mathcal{C}^{\mathrm{op}} \to \mathbf{Cat}$ 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* (3.2) *is invertible.* In this case, it is even clearer that extensivity coincides with the well-known notion of preservation of binary products.

Recall that *preservation of binary products does imply preservation of preterminal objects.* Indeed, an object X is a preterminal object if and only if the projection $\pi_X : X \times X \to X$ is invertible. Hence, since a binary product preserving functor should preserve the projections, we get the result. Lemma 3.5 is the appropriate analogue of this observation suitably applied to the context of extensive indexed categories.*

Lemma 3.5 (Preservation of terminal objects)**.** *Let* $\mathcal{L} : \mathcal{C}^{\mathrm{op}} \to \mathbf{Cat}$ *be an extensive indexed category which is not (naturally isomorphic to the functor) constantly equal to* $\mathbb{0}$*. The unique functor*

$$\mathcal{L}(\mathbb{0}) \to \mathbb{1} \tag{3.5}$$

is an equivalence. If, furthermore, (3.2) *is an isomorphism, then* (3.5) *is invertible.*

*On the one hand, as mentioned above, Lemma 3.5 is actually part of a general fact: if a functor preserves binary (bicategorical) products, then it preserves preterminal objects as well (see, for instance, [28, Remark 4.5]). On the other hand, seeing our extensivity property as a generalization of that of *extensive categories*, the reader might want to compare Lemma 3.5 with [10, Proposition 2.8].

Proof: Firstly, given any $X \in \mathcal{C}$ such that $\mathcal{L}(X)$ is not (isomorphic to) the initial object of **Cat**, we have that $\mathcal{L}(i_X : \mathbb{0} \to X)$ 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 $\iota_{\mathbb{0}} : \mathbb{0} \to \mathbb{0} \sqcup \mathbb{0}$ is an isomorphism, $(\mathcal{L}(\iota_{\mathbb{0}}), \mathcal{L}(\iota_{\mathbb{0}}))$ is an equivalence and

$$\begin{tikzcd} \mathcal{L}(\mathbb{0} \sqcup \mathbb{0}) \arrow[r, "(\mathcal{L}(\iota_{\mathbb{0}}),\mathcal{L}(\iota_{\mathbb{0}}))"] \arrow[rr, bend right, "\mathcal{L}(\iota_{\mathbb{0}})"'] & \mathcal{L}(\mathbb{0}) \times \mathcal{L}(\mathbb{0}) \arrow[r, "\pi_{\mathcal{L}(\mathbb{0})}"] & \mathcal{L}(\mathbb{0}), \end{tikzcd} \tag{3.6}$$

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

Now, we proceed to study the cocartesian structure of $\Sigma_{\mathcal{C}}\mathcal{L}$. In order to do so, we show in Theorem 3.3 that, in the case of extensive indexed categories, the hypothesis of Proposition C.1 always holds.

Theorem 3.3. *Let $\mathcal{L} : \mathcal{C}^{\mathrm{op}} \to$ **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*

$$\begin{tikzcd} \mathcal{L}(W \sqcup X) \arrow[r, bend right, "\mathcal{L}(\iota_W)"'] \arrow[r, phantom, "\perp"] & \mathcal{L}(W) \arrow[l, bend right, "\mathcal{S}^{(W,X)} \circ \left(\mathrm{id}_{\mathcal{L}(W)}, \mathbb{0}\right)"'] \end{tikzcd} \tag{3.7}$$

in which, by abuse of language, $\mathbb{0} : \mathcal{L}(W) \to \mathcal{L}(X)$ is the functor constantly equal to $\mathbb{0}$. Dually, we have an adjunction

$$\begin{tikzcd} \mathcal{L}(W) \arrow[r, bend right, "\mathcal{S}^{(W,X)} \circ \left(\mathrm{id}_{\mathcal{L}(W)}, \mathbb{1}\right)"'] \arrow[r, phantom, "\perp"] & \mathcal{L}(W \sqcup X) \arrow[l, bend right, "\mathcal{L}(\iota_W)"'] \end{tikzcd} \tag{3.8}$$

provided that $\mathcal{L}(X)$ has terminal object $\mathbb{1}$ and, by abuse of language, we denote by $\mathbb{1} : \mathcal{L}(W) \to \mathcal{L}(X)$ the functor constantly equal to $\mathbb{1}$.

Proof: Assuming that $\mathcal{L}(X)$ has initial object $\mathbb{0}$, we have the adjunction

$$\begin{tikzcd} \mathcal{L}(W) \times \mathcal{L}(X) \arrow[rr, bend right, "\pi_{\mathcal{L}(W)}"'] \arrow[rr, phantom, "\perp"] & & \mathcal{L}(W) \arrow[ll, bend right, "\left(\mathrm{id}_{\mathcal{L}(W)}, \mathbb{0}\right)"'] \end{tikzcd} \tag{3.9}$$

whose unit is the identity and counit is pointwise given by $\varepsilon_{(w,x)} = (\mathrm{id}_w, \mathbb{0} \to x)$. Therefore we have the composition of adjunctions

$$\begin{tikzcd} \mathcal{L}(W \sqcup X) \arrow[rr, bend right, "(\mathcal{L}(\iota_W),\mathcal{L}(\iota_X))"'] \arrow[rr, phantom, "\perp"] \arrow[rrrr, bend right=40, "\mathcal{L}(\iota_W)"'] & & \mathcal{L}(W) \times \mathcal{L}(X) \arrow[ll, bend right, "\mathcal{S}^{(W,X)}"'] \arrow[rr, bend right, "\pi_{\mathcal{L}(W)}"'] \arrow[rr, phantom, "\perp"] & & \mathcal{L}(W). \arrow[ll, bend right, "\left(\mathrm{id}_{\mathcal{L}(W)}, \mathbb{0}\right)"'] \arrow[llll, bend right=40, "\mathcal{S}^{(W,X)} \circ \left(\mathrm{id}_{\mathcal{L}(W)}, \mathbb{0}\right)"'] \end{tikzcd}$$

■

Corollary 3.6 (Cocartesian structure of $\Sigma_{\mathcal{C}}\mathcal{L}$)**.** *Let $\mathcal{L} : \mathcal{C}^{\mathrm{op}} \to \mathbf{Cat}$ be an extensive strictly indexed category, with initial objects $\mathbb{0} \in \mathcal{L}(W)$ for each $W \in \mathcal{C}$. In this case, the category $\Sigma_{\mathcal{C}}\mathcal{L}$ has initial object $\mathbb{0} = (\mathbb{0}, \mathbb{0}) \in \Sigma_{\mathcal{C}}\mathcal{L}$, and (fibred) binary coproduct given by $(W, w) \sqcup (X, x) = \left(W \sqcup X, \mathcal{S}^{(W,X)}(w, x)\right)$.*

Proof: In fact, by Proposition C.1, we have that $(\mathbb{0}, \mathbb{0})$ is the initial object of $\Sigma_{\mathcal{C}}\mathcal{L}$. Moreover, given $((W, w), (X, x)) \in \Sigma_{\mathcal{C}}\mathcal{L} \times \Sigma_{\mathcal{C}}\mathcal{L}$, we have that

$$\mathcal{S}^{(W,X)} \circ \left(\mathrm{id}_{\mathcal{L}(W)}, \mathbb{0}\right) = \mathcal{L}(\iota_W)! \;\dashv\; \mathcal{L}(\iota_W)$$
$$\mathcal{S}^{(W,X)} \circ \left(\mathbb{0}, \mathrm{id}_{\mathcal{L}(X)}\right) = \mathcal{L}(\iota_X)! \;\dashv\; \mathcal{L}(\iota_X)$$

by Theorem 3.3. Therefore we get that

$$\begin{aligned}
&\left(W \sqcup X, \mathcal{S}^{(W,X)}(w, x)\right) \\
&\cong \left(W \sqcup X, \mathcal{S}^{(W,X)}(w, \mathbb{0}) \sqcup \mathcal{S}^{(W,X)}(\mathbb{0}, x)\right) && \{\ \mathcal{S}^{(W,X)} \text{ preserves coproducts}\ \} \\
&\cong \left(W \sqcup X, \mathcal{S}^{(W,X)} \circ \left(\mathrm{id}_{\mathcal{L}(W)}, \mathbb{0}\right)(w) \sqcup \mathcal{S}^{(W,X)} \circ \left(\mathbb{0}, \mathrm{id}_{\mathcal{L}(X)}\right)(x)\right) \\
&\cong (W \sqcup X, \mathcal{L}(\iota_W)!(w) \sqcup \mathcal{L}(\iota_X)!(x)) && \{\ \text{Theorem 3.3}\ \} \\
&\cong (W, w) \sqcup (X, x). && \{\ \text{Proposition C.1}\ \}
\end{aligned}$$

■

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

Codually, we have:

Corollary 3.7 (Cocartesian structure of $\Sigma_{\mathcal{C}}\mathcal{L}^{\mathrm{op}}$)**.** *Let $\mathcal{L} : \mathcal{C}^{\mathrm{op}} \to \mathbf{Cat}$ 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 $\Sigma_{\mathcal{C}}\mathcal{L}^{\mathrm{op}}$ has (fibred) initial object $\mathbb{0} = (\mathbb{0}, \mathbb{1}) \in \Sigma_{\mathcal{C}}\mathcal{L}^{\mathrm{op}}$, and (fibred) binary coproduct given by*

$$(W, w) \sqcup (X, x) = \Big(W \sqcup X, \mathcal{S}^{(W,X)}(w, x)\Big)\,. \tag{3.10}$$

Definition 3.8 (Σ-bimodel for sum types)**.** A strictly indexed category $\mathcal{L} : \mathcal{C}^{\mathrm{op}} \to \mathbf{Cat}$ 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.

3.6. Distributive property. It is clear that $\Sigma_{\mathcal{C}}\mathcal{L}$ is bi-Cartesian closed provided that $\mathcal{L} : \mathcal{C}^{\mathrm{op}} \to \mathbf{Cat}$ is Σ-bimodel for function types and sum types. Therefore, in this case, 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 3.4.

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

$$\left\langle W \times \iota_Y^{Y \sqcup Z}, W \times \iota_Z^{Y \sqcup Z}\right\rangle : (W \times Y) \sqcup (W \times Z) \to W \times (Y \sqcup Z)\,, \tag{3.11}$$

induced by $W \times \mathcal{L}(\iota_Y)$ and $W \times \mathcal{L}(\iota_Z)$, 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 diagrams

$$\begin{tikzcd}
W \times (Y \sqcup Z) \arrow[drr, "\pi_W^{W\times(Y\sqcup Z)}"] & & \\
(W \times Y) \sqcup (W \times Z) \arrow[u, "\langle W \times \iota_Y, W \times \iota_Z \rangle", "\cong"'] \arrow[rr, "\left\langle \pi_W^{W\times Y}, \pi_W^{W\times Z}\right\rangle"'] & & W
\end{tikzcd}$$

$$
\begin{tikzcd}
W \times (Y \sqcup Z) \arrow[drr, "\pi^{W\times(Y\sqcup Z)}_{(Y\sqcup Z)}"] & & \\
(W \times Y) \sqcup (W \times Z) \arrow[u, "\langle W\times\iota_Y, W\times\iota_Z\rangle", "\cong"'] \arrow[rr, "\left\langle \iota_Y\circ\pi_Y^{W\times Y}, \iota_Z\circ\pi_Z^{W\times Z}\right\rangle", "\pi_Y^{W\times Y}\sqcup\pi_Z^{W\times Z}"'] & & (Y \sqcup Z)
\end{tikzcd}
$$

commute. Therefore we have:

Lemma 3.9. *Let $\mathcal{L} : \mathcal{C}^{\mathrm{op}} \to \mathbf{Cat}$ 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*

$$
\begin{tikzcd}
\mathcal{L}(W \times (Y \sqcup Z)) \arrow[d, "\mathcal{L}(\langle W\times\iota_Y, W\times\iota_Z\rangle)"', "\cong"] & & \\
\mathcal{L}((W \times Y) \sqcup (W \times Z)) \arrow[d, bend left, "(\mathcal{L}(\iota_{W\times Y}),\mathcal{L}(\iota_{W\times Z}))"] & & \mathcal{L}(W) \arrow[ll, "\mathcal{L}\left(\left\langle \pi_W^{W\times Y}, \pi_W^{W\times Z}\right\rangle\right)"'] \arrow[ull, bend right, "\mathcal{L}\left(\pi_W^{W\times(Y\sqcup Z)}\right)"'] \arrow[dll, bend left, "\left(\mathcal{L}\left(\pi_W^{W\times Y}\right),\mathcal{L}\left(\pi_W^{W\times Z}\right)\right)"] \\
\mathcal{L}(W \times Y) \times \mathcal{L}(W \times Z) \arrow[u, bend left, "\mathcal{S}^{(W\times Y, W\times Z)}"] & &
\end{tikzcd}
\tag{3.12}
$$

$$
\begin{tikzcd}
\mathcal{L}(W \times (Y \sqcup Z)) \arrow[d, "\mathcal{L}(\langle W\times\iota_Y, W\times\iota_Z\rangle)"', "\cong"] & & \\
\mathcal{L}((W \times Y) \sqcup (W \times Z)) \arrow[d, bend left, "(\mathcal{L}(\iota_{W\times Y}),\mathcal{L}(\iota_{W\times Z}))"] & & \mathcal{L}(Y \sqcup Z) \arrow[ll, "\mathcal{L}\left(\left\langle \pi_Y^{W\times Y}, \pi_Z^{W\times Z}\right\rangle\right)"'] \arrow[ull, bend right, "\mathcal{L}\left(\pi_{(Y\sqcup Z)}^{W\times(Y\sqcup Z)}\right)"'] \arrow[d, bend right, "(\mathcal{L}(\iota_Y),\mathcal{L}(\iota_Z))"'] \\
\mathcal{L}(W \times Y) \times \mathcal{L}(W \times Z) \arrow[u, bend left, "\mathcal{S}^{(W\times Y, W\times Z)}"] & & \mathcal{L}(Y) \times \mathcal{L}(Z) \arrow[ll, "\mathcal{L}\left(\pi_Y^{W\times Y}\right)\times\mathcal{L}\left(\pi_Z^{W\times Z}\right)"] \arrow[u, bend right, "\mathcal{S}^{(Y,Z)}"']
\end{tikzcd}
\tag{3.13}
$$

commute.

Theorem 3.4. *Let* $\mathcal{L} : \mathcal{C}^{\mathrm{op}} \to \mathbf{Cat}$ *be* Σ*-bimodel for sum and tuple types, in which* $\mathcal{C}$ *is a distributive category. The category* $\Sigma_{\mathcal{C}}\mathcal{L}$ *is a distributive category.*

Proof: By Proposition 3.1 and Corollary 3.6, 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 (3.11) are invertible. However, by [23, 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 $((W,w),(Y,y),(Z,z)) \in \Sigma_{\mathcal{C}}\mathcal{L} \times \Sigma_{\mathcal{C}}\mathcal{L} \times \Sigma_{\mathcal{C}}\mathcal{L}$

$$
\begin{aligned}
&(W,w) \times ((Y,y) \sqcup (Z,z)) \\
&\cong (W,w) \times \left(Y \sqcup Z, \mathcal{S}^{(Y,Z)}(y,z)\right) && \{ \text{ Corollary 3.6 } \} \\
&\cong \left(W \times (Y \sqcup Z), \mathcal{L}(\pi_W)(w) \times \mathcal{L}(\pi_{Y \sqcup Z})\mathcal{S}^{(Y,Z)}(y,z)\right), && \{ \text{ Proposition 3.1 } \}
\end{aligned}
$$

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

$$
\left((W \times Y) \sqcup (W \times Z), \mathcal{L}\left(\langle W \times \iota_Y, W \times \iota_Z \rangle\right)\left(\mathcal{L}(\pi_W)(w) \times \mathcal{L}(\pi_{Y \sqcup Z})\mathcal{S}^{(Y,Z)}(y,z)\right)\right). \tag{3.14}
$$

Moreover, we have the natural isomorphisms

$$
\begin{aligned}
&\mathcal{L}(\langle W \times \iota_Y, W \times \iota_Z \rangle)\left(\mathcal{L}(\pi_W)(w) \times \mathcal{L}(\pi_{Y \sqcup Z})\mathcal{S}^{(Y,Z)}(y,z)\right) \\
&\cong \mathcal{L}(\langle W \times \iota_Y, W \times \iota_Z \rangle)(\mathcal{L}(\pi_W)(w)) \times \mathcal{L}(\langle W \times \iota_Y, W \times \iota_Z \rangle)\left(\mathcal{L}(\pi_{Y \sqcup Z})\mathcal{S}^{(Y,Z)}(y,z)\right) && \{ \mathcal{L}\left(\langle W \times \iota_Y, W \times \iota_Z \rangle\right) \text{ invertible } \} \\
&= \mathcal{S}^{(W \times Y, W \times Z)}(\mathcal{L}(\pi_W)(w), \mathcal{L}(\pi_W)(w)) \times \mathcal{L}(\langle W \times \iota_Y, W \times \iota_Z \rangle) \circ \mathcal{L}(\pi_{Y \sqcup Z}) \circ \mathcal{S}^{(Y,Z)}(y,z) && \{ \text{ Diagram (3.12) } \} \\
&= \mathcal{S}^{(W \times Y, W \times Z)}(\mathcal{L}(\pi_W)(w), \mathcal{L}(\pi_W)(w)) \times \mathcal{S}^{(W \times Y, W \times Z)}(\mathcal{L}(\pi_Y)(y), \mathcal{L}(\pi_Z)(z)), && \{ \text{ Diagram (3.13) } \}
\end{aligned}
$$

which is naturally isomorphic to

$$
\mathcal{S}^{(W \times Y, W \times Z)}\left(\mathcal{L}(\pi_W)(w) \times \mathcal{L}(\pi_Y)(y), \mathcal{L}(\pi_W)(w) \times \mathcal{L}(\pi_Z)(z)\right). \tag{3.15}
$$

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

$$
\begin{aligned}
&(W,w) \times ((Y,y) \sqcup (Z,z)) \\
&\cong \left((W \times Y) \sqcup (W \times Z), \mathcal{L}(\langle W \times \iota_Y, W \times \iota_Z \rangle)\left(\mathcal{L}(\pi_W)(w) \times \mathcal{L}(\pi_{Y \sqcup Z})\mathcal{S}^{(Y,Z)}(y,z)\right)\right) && \{ \text{ Eq. (3.14) } \} \\
&\cong \left((W \times Y) \sqcup (W \times Z), \mathcal{S}^{(W \times Y, W \times Z)}(\mathcal{L}(\pi_W)(w) \times \mathcal{L}(\pi_Y)(y), \mathcal{L}(\pi_W)(w) \times \mathcal{L}(\pi_Z)(z))\right) && \{ \text{ Eq. (3.15) } \}
\end{aligned}
$$

$$\cong (W\times Y, \mathcal{L}(\pi_W)(w)\times\mathcal{L}(\pi_Y)(y)) \sqcup (W\times Z, \mathcal{L}(\pi_W)(w)\times\mathcal{L}(\pi_Z)(z)) \qquad \left\{ \text{ Corollary 3.6 } \right\}$$
$$((W,w)\times(Y,y))\sqcup((W,w)\times(Z,z)), \qquad \left\{ \text{Proposition 3.1} \right\}$$

which completes our proof. ∎

Codually, we have:

Theorem 3.5. *Let $\mathcal{L} : \mathcal{C}^{\mathrm{op}} \to \mathbf{Cat}$ be a Σ-bimodel for sum and tuple types, in which $\mathcal{C}$ is a distributive category. Then we conclude that $\Sigma_{\mathcal{C}}\mathcal{L}^{\mathrm{op}}$ is a distributive category.*

3.7. Distributive and extensive properties. Recall that $\mathcal{C}$ is extensive if the basic indexed category

$$\mathcal{C}/- : \mathcal{C}^{\mathrm{op}} \to \mathbf{Cat}$$

is extensive (see [10, Definition 2.1]). Since *free cocompletions under (finite)*[*] *coproducts* and *free distributive categories*[†] are extensive, categorical models for variant types are usually extensive.

Recall that *extensive categories with finite products are distributive*[‡], so, assuming that $\mathcal{C}$ is extensive (which we claim to not be a wild assumption for our context), the following result is a generalization of Thm. 3.4.

Theorem 3.6. *Let $\mathcal{L} : \mathcal{C}^{\mathrm{op}} \to \mathbf{Cat}$ be an extensive strictly indexed category, in which $\mathcal{C}$ is a extensive category. Assume that we have initial objects $\mathbb{0} \in \mathcal{L}(W)$. In this case, the category $\Sigma_{\mathcal{C}}\mathcal{L}$ is extensive.*

Proof: We denote by $\mathcal{S}_{\mathcal{L}}^{(W,X)} : \mathcal{L}(W)\times\mathcal{L}(X) \to \mathcal{L}(W\sqcup X)$ 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 3.6. 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 indded have that

$$\mathcal{S}_{\Sigma_{\mathcal{C}}\mathcal{L}/-}^{((W,w),(X,x))} : \Sigma_{\mathcal{C}}\mathcal{L}/(W,w) \times \Sigma_{\mathcal{C}}\mathcal{L}/(X,x) \to \Sigma_{\mathcal{C}}\mathcal{L}/\left((W,w)\sqcup(X,x)\right) \tag{3.16}$$

defined by the coproduct of the morphisms is an equivalence. Explicitly, given objects $A = ((W_0,w_0),(f : W_0 \to W, f' : w_0 \to \mathcal{L}(f)\,w))$ of $\Sigma_{\mathcal{C}}\mathcal{L}/(W,w)$ and

[*] The proof given for [10, Proposition 2.4] also applies to the finite case.

[†] See [10, Proposition 3.6].

[‡] See [10, Proposition 4.5].

[§] The result also holds for the non-strict scenario.

$B = ((X_0, x_0), (g : X_0 \to X, g' : x_0 \to \mathcal{L}(g)\, x))$, $\mathcal{S}^{((W,w),(X,x))}_{\Sigma_{\mathcal{C}}\mathcal{L}/-}(A, B)$ is given by

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

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 $\mathcal{C}/W \times \mathcal{C}/X \to \mathcal{C}/W \sqcup X$. ∎

Theorem 3.7. *Let $\mathcal{L} : \mathcal{C}^{\mathrm{op}} \to \mathbf{Cat}$ be an extensive strictly indexed category, in which $\mathcal{C}$ is a extensive category. Assume that we have terminal objects $\mathbb{1} \in \mathcal{L}(W)$. In this case, the category $\Sigma_{\mathcal{C}}\mathcal{L}^{\mathrm{op}}$ is extensive.*

3.8. $\mu\nu$-polynomials. We examine the existence of *$\mu\nu$-polynomials* in $\Sigma_{\mathcal{C}}\mathcal{L}$ and $\Sigma_{\mathcal{C}}\mathcal{L}^{\mathrm{op}}$. In order to do so, we employ the results and terminology established in Appendices F and G. We also need the following definitions.

Definition 3.10 ($\mu\nu\mathsf{Poly}_{\mathcal{L}}$)**.** Let $\mathcal{C}$ be a category with $\mu\nu$-polynomials, and $\mathcal{L} : \mathcal{C}^{\mathrm{op}} \to \mathbf{Cat}$ an extensive strictly indexed category with finite biproducts. We define the category $\mu\nu\mathsf{Poly}_{\mathcal{L}}$ as the smallest subcategory of $\mathbf{Cat}$ satisfying the following.

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

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

    of the extensive structure (see (3.4)) is a morphism of $\mu\nu\mathsf{Poly}_{\mathcal{L}}$;

M6. given an object $\mathcal{D}$ of $\mu\nu\mathsf{Poly}_{\mathcal{C}}$, a morphism $\overline{H} : \mathcal{D} \times \mathcal{C} \to \mathcal{C}$ of $\mu\nu\mathsf{Poly}_{\mathcal{C}}$ and any object $X \in \mathcal{D}'$,

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

are morphisms of $\mu\nu\mathsf{Poly}_{\mathcal{L}}$;

M7. for each $(W, X) \in \mathcal{C} \times \mathcal{C}$, the functors induced by the projections

$$\mathcal{L}(\pi_1) : \mathcal{L}(W) \to \mathcal{L}(W \times X), \qquad \mathcal{L}(\pi_2) : \mathcal{L}(X) \to \mathcal{L}(W \times X)$$

are morphisms of $\mu\nu\mathsf{Poly}_{\mathcal{L}}$;

M8. if $E : \mathcal{D} \to \mathcal{D}'$ and $J : \mathcal{D} \to \mathcal{D}''$ are morphisms of $\mu\nu\mathsf{Poly}_{\mathcal{L}}$, then so is $(E, J) : \mathcal{D} \to \mathcal{D}' \times \mathcal{D}''$;

M9. if $\mathcal{D}', \mathcal{D}$ are objects of $\mu\nu\mathsf{Poly}_{\mathcal{L}}$, $h : \mathcal{D}' \times \mathcal{D} \to \mathcal{D}$ is a morphism of $\mu\nu\mathsf{Poly}_{\mathcal{L}}$ and $\mu h : \mathcal{D}' \to \mathcal{D}$ exists, then μh is a morphism of $\mu\nu\mathsf{Poly}_{\mathcal{L}}$;

M10. if $\mathcal{D}', \mathcal{D}$ are objects of $\mu\nu\mathsf{Poly}_{\mathcal{L}}$, $h : \mathcal{D}' \times \mathcal{D} \to \mathcal{D}$ is a morphism of $\mu\nu\mathsf{Poly}_{\mathcal{L}}$ and $\nu h : \mathcal{D}' \to \mathcal{D}$ exists, then νh is a morphism of $\mu\nu\mathsf{Poly}_{\mathcal{L}}$.

Recall that a strictly indexed category $\mathcal{L} : \mathcal{C}^{\mathrm{op}} \to \mathbf{Cat}$ respects terminal coalgebras and initial algebras if, for any morphism f of $\mathcal{C}$, $\mathcal{L}(f)$ preserves terminal coalgebras and initial algebras.[*]

Definition 3.11 (Σ-bimodel for inductive and coinductive types)**.** We say that $\mathcal{L} : \mathcal{C}^{\mathrm{op}} \to \mathbf{Cat}$ is a Σ-*bimodel for inductive and coinductive types* if:

$*1)$ $\mathcal{C}$ has $\mu\nu$-polynomials;
$*2)$ $\mathcal{L}$ is a strictly indexed category;
$*3)$ $\mathcal{L} : \mathcal{C}^{\mathrm{op}} \to \mathbf{Cat}$ has indexed biproducts, denoted by $+$ with zero object denoted by $\mathbb{1} = \mathbb{0}$;
$*4)$ $\mathcal{L}$ is extensive;
$*5)$ whenever $\mathcal{D}$ is an object of $\mu\nu\mathsf{Poly}_{\mathcal{L}}$ and $e : \mathcal{D} \to \mathcal{D}$ is a morphism of $\mu\nu\mathsf{Poly}_{\mathcal{L}}$, μe and νe exist;
$*6)$ $\mathcal{L}$ respects terminal coalgebras and initial algebras.

For short, in this section, such an indexed category is called a $*$-*indexed category*.

[*]See Definitions E.5, E.8, and E.9.

Lemma 3.12. *Let $\mathcal{L} : \mathcal{C}^{\mathrm{op}} \to \mathbf{Cat}$ be a $*$-indexed category. If $\mathcal{D}, \mathcal{D}'$ are objects of $\mu\nu\mathsf{Poly}_{\mathcal{L}}$ then, whenever $h : \mathcal{D}' \times \mathcal{D} \to \mathcal{D}$ is a morphism of $\mu\nu\mathsf{Poly}_{\mathcal{L}}$,*

$$\mu h : \mathcal{D}' \to \mathcal{D} \qquad \textit{and} \qquad \nu h : \mathcal{D}' \to \mathcal{D}$$

exist.

Proof: By Proposition D.1, it is enough to show that, for each $x \in \mathcal{D}'$, μh^x and νh^x exist.

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

$$\begin{tikzcd} \mathcal{D} \arrow[r, "(1,\mathrm{id}_{\mathcal{D}})"] \arrow[rrr, bend right, "h^x"'] & \mathbb{1} \times \mathcal{D} \arrow[r, "(x\circ\pi_1,\mathrm{id}_{\mathcal{D}}\circ\pi_2)"] & \mathcal{D}' \times \mathcal{D} \arrow[r, "h"] & \mathcal{D} \end{tikzcd}$$

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

Definition 3.13 ($\mu\nu\mathcal{L}$-indexed category and indexed functor)**.** Let $\mathcal{L} : \mathcal{C}^{\mathrm{op}} \to \mathbf{Cat}$, $\mathcal{L}' : \mathcal{D}^{\mathrm{op}} \to \mathbf{Cat}$ 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 $\mu\nu\mathsf{Poly}_{\mathcal{C}}$;
$\mu\nu\mathcal{L}2$) $\mathcal{L}'(W)$ is an object of $\mu\nu\mathsf{Poly}_{\mathcal{L}}$ for any W in $\mathcal{D}$.

A strictly indexed functor $(\overline{H}, h)$ between $\mathcal{L}' : \mathcal{D}^{\mathrm{op}} \to \mathbf{Cat}$ and $\mathcal{L}'' : \mathcal{E}^{\mathrm{op}} \to \mathbf{Cat}$ 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;
$\mu\nu\mathcal{L}4$) $\overline{H} : \mathcal{D} \to \mathcal{E}$ is a morphism of $\mu\nu\mathsf{Poly}_{\mathcal{C}}$;
$\mu\nu\mathcal{L}5$) for each $X \in \mathcal{D}$, $h_X : \mathcal{L}'(X) \to \mathcal{L}'' \circ \overline{H}(X)$ is a morphism of $\mu\nu\mathsf{Poly}_{\mathcal{L}}$.

Theorem 3.8. *Let $\mathcal{L}' : \mathcal{D}^{\mathrm{op}} \to \mathbf{Cat}$ be a strictly indexed category and $\mathcal{L} : \mathcal{C}^{\mathrm{op}} \to \mathbf{Cat}$ a $*$-indexed category. Assume that $(\overline{H}, h)$ is a $\mu\nu\mathcal{L}$-indexed functor, and $H : \Sigma_{\mathcal{E}\times\mathcal{D}}(\mathcal{L}' \underline{\times} \mathcal{L}) \cong (\Sigma_{\mathcal{E}}\mathcal{L}') \times (\Sigma_{\mathcal{D}}\mathcal{L}) \to \Sigma_{\mathcal{D}}\mathcal{L}$ is the corresponding split fibration functor. We have that:*

– *$\mu H : \Sigma_{\mathcal{E}}\mathcal{L}' \to \Sigma_{\mathcal{D}}\mathcal{L}$ exists and is the split fibration functor induced by the $\mu\nu\mathcal{L}$-indexed functor*

$$\left(\mu\overline{H} : \mathcal{E} \to \mathcal{D},\ \mu\left(\underline{h}_{(-)}\right)\right) \tag{3.17}$$

in which

$$\mu\left(\underline{h}_{(X)}\right) = \mu\underline{h}_X = \mu\left(\mathcal{L}(\mathfrak{in}_{\overline{H}^X})^{-1}h_{\left(X,\mu\overline{H}^X\right)}\right) : \mathcal{L}'(X) \to \mathcal{L}(\mu\overline{H}^X). \qquad (3.18)$$

– $\nu H : \Sigma_{\mathcal{E}}\mathcal{L}' \to \Sigma_{\mathcal{D}}\mathcal{L}$ *exists and is the split fibration functor induced by the* $\mu\nu\mathcal{L}$*-indexed functor*

$$\left(\nu\overline{H} : \mathcal{E} \to \mathcal{D},\, \nu\left(\overline{h}_{(-)}\right)\right) \qquad (3.19)$$

in which

$$\nu\left(\overline{h}_{(X)}\right) = \nu\overline{h}_X = \nu\left(\mathcal{L}(\mathsf{out}_{\overline{H}^X})h_{\left(X,\nu\overline{H}^X\right)}\right) : \mathcal{L}''(X) \to \mathcal{L}'(\nu\overline{H}^X). \qquad (3.20)$$

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 $\mu\nu\mathsf{Poly}_{\mathcal{C}}$ and $\overline{H}$ is a morphism of $\mu\nu\mathsf{Poly}_{\mathcal{C}}$, we have that $\mu\overline{H}$ and $\nu\overline{H}$ exist by Lemma 1.2 (and, hence, are morphisms in $\mu\nu\mathsf{Poly}_{\mathcal{C}}$). Moreover, we have that $\mathcal{L}(\mathsf{out}_{\overline{H}^X})$ and $\mathcal{L}(\mathfrak{in}_{\overline{H}^X})^{-1}$ are morphisms of $\mu\nu\mathsf{Poly}_{\mathcal{L}}$ by M6. of Definition 3.10.

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

$$\begin{aligned} h_{\left(X,\mu\overline{H}^X\right)} :\ & \mathcal{L}'(X) \times \mathcal{L}\left(\mu\overline{H}^X\right) \to \mathcal{L}\circ\overline{H}\left(X,\mu\overline{H}^X\right) \\ h_{\left(X,\nu\overline{H}^X\right)} :\ & \mathcal{L}'(X) \times \mathcal{L}\left(\nu\overline{H}^X\right) \to \mathcal{L}\circ\overline{H}\left(X,\nu\overline{H}^X\right) \end{aligned}$$

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

We conclude, then, that the compositions

$$\begin{aligned} \underline{h}_X = \mathcal{L}(\mathfrak{in}_{\overline{H}^X})^{-1}h_{\left(X,\mu\overline{H}^X\right)} :\ & \mathcal{L}'(X) \times \mathcal{L}\left(\mu\overline{H}^X\right) \to \mathcal{L}\left(\mu\overline{H}^X\right) \\ \overline{h}_X = \mathcal{L}(\mathsf{out}_{\overline{H}^X})h_{\left(X,\nu\overline{H}^X\right)} :\ & \mathcal{L}'(X) \times \mathcal{L}\left(\nu\overline{H}^X\right) \to \mathcal{L}\left(\nu\overline{H}^X\right) \end{aligned}$$

are also morphisms of $\mu\nu\mathsf{Poly}_{\mathcal{L}}$. Thus, we have that $\mu\underline{h}_X$ and $\nu\overline{h}_X$ exist (and are morphisms of $\mu\nu\mathsf{Poly}_{\mathcal{L}}$) by Lemma 3.12.

Finally, since $\mathcal{L}$ respects initial algebras and terminal coalgebras, we have that $(\overline{H}, h)$ satisfies the hypotheses of Corollary F.1 and Theorem G.2. Therefore μH and νH exist and are induced by (3.17) and (3.19) respectively.

The fact that (3.17) and (3.19) 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, $\mu\overline{H}$ is a morphism of $\mu\nu\mathsf{Poly}_{\mathcal{C}}$ (as observed above) and $\mu\underline{h}_X, \nu\overline{h}_X$ are morphisms of $\mu\nu\mathsf{Poly}_{\mathcal{L}}$ (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 3.9, our main theorem about $\mu\nu$-polynomials in $\Sigma_{\mathcal{C}}\mathcal{L}$, we prove Lemma 3.16 which establishes a bijection between objects of $\mu\nu\mathsf{Poly}_{\Sigma_{\mathcal{C}}\mathcal{L}}$ and indexed categories.

Definition 3.14. Let $\mathcal{L} : \mathcal{C}^{\mathrm{op}} \to \mathbf{Cat}$ 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 $\mathbb{1} : \mathbb{1} \to \mathbf{Cat}$ 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 3.15. *Let $\mathcal{L} : \mathcal{C}^{\mathrm{op}} \to \mathbf{Cat}$ 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 $\mathbb{1} : \mathbb{1} \to \mathbf{Cat}$ is a $\mu\nu\mathcal{L}$-indexed category since $\mathbb{1} \in \mu\nu\mathsf{Poly}_{\mathcal{C}}$ and $\mathbb{1} \in \mu\nu\mathsf{Poly}_{\mathcal{L}}$. Furthermore, $\mathcal{L} : \mathcal{C}^{\mathrm{op}} \to \mathbf{Cat}$ is a $\mu\nu\mathcal{L}$-indexed category by the definition of $\mu\nu\mathsf{Poly}_{\mathcal{L}}$.

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

- we have that $(\mathcal{D}, \mathcal{E}) \in \mu\nu\mathsf{Poly}_{\mathcal{C}} \times \mu\nu\mathsf{Poly}_{\mathcal{C}}$. Thus
$$(\mathcal{D} \times \mathcal{E}) \in \mu\nu\mathsf{Poly}_{\mathcal{C}}; \tag{3.21}$$
- for any $(W, W') \in \mathcal{D}\times\mathcal{E}$, the categories $\mathcal{L}'(W)$ and $\mathcal{L}''(W')$ are objects of $\mu\nu\mathsf{Poly}_{\mathcal{L}}$. Thus
$$\mathcal{L}' \underline{\times} \mathcal{L}''\,(W, W') = \mathcal{L}'(W) \times \mathcal{L}''(W') \in \mu\nu\mathsf{Poly}_{\mathcal{L}}. \tag{3.22}$$

By (3.21) and (3.22), we conclude that $\mathcal{L}' \underline{\times} \mathcal{L}'' : (\mathcal{D} \times \mathcal{E})^{\mathrm{op}} \to \mathbf{Cat}$ is a $\mu\nu\mathcal{L}$-indexed category. ■

Lemma 3.16. *Let $\mathcal{L} : \mathcal{C}^{\mathrm{op}} \to \mathbf{Cat}$ be a strictly indexed category. We define a bijection $\overline{\partial}$ between the set of objects of $\mu\nu\mathsf{Poly}_{\Sigma_{\mathcal{C}}\mathcal{L}}$ and $\overline{\underline{\times}}\mathcal{L}$.*

Proof: We define the bijection $\overline{\partial} : \mathrm{obj}\left(\mu\nu\mathsf{Poly}_{\Sigma_{\mathcal{C}}\mathcal{L}}\right) \to \overline{\underline{\times}}\mathcal{L}$ inductively as follows:

$\overline{\partial}$1. terminal respecting: $\overline{\partial}\,(\mathbb{1}) \coloneqq (\mathbb{1} : \mathbb{1} \to \mathbf{Cat})$;
$\overline{\partial}$2. basic element: $\overline{\partial}\,(\Sigma_{\mathcal{C}}\mathcal{L}) \coloneqq (\mathcal{L} : \mathcal{C}^{\mathrm{op}} \to \mathbf{Cat})$;

$\overline{\partial}$3. product respecting: given $(\mathcal{D}, \mathcal{D}') \in \mu\nu\mathsf{Poly}_{\Sigma_\mathcal{C}\mathcal{L}} \times \mu\nu\mathsf{Poly}_{\Sigma_\mathcal{C}\mathcal{L}}$,

$$\overline{\partial}\left(\mathcal{D} \times \mathcal{D}'\right) \coloneqq \overline{\partial}\left(\mathcal{D}\right) \underline{\times} \overline{\partial}\left(\mathcal{D}'\right).$$

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

$\overline{\Sigma}$1. terminal respecting: $\overline{\Sigma}\left(\mathbb{1} : \mathbb{1} \to \mathbf{Cat}\right) \coloneqq \mathbb{1}$;
$\overline{\Sigma}$2. basic element: $\overline{\Sigma}\left(\mathcal{L} : \mathcal{C}^{\mathrm{op}} \to \mathbf{Cat}\right) \coloneqq \Sigma_\mathcal{C}\mathcal{L}$;
$\overline{\Sigma}$3. product respecting: given $\left(\mathcal{L}' : \mathcal{D}^{\mathrm{op}} \to \mathbf{Cat}, \mathcal{L}'' : \mathcal{E}^{\mathrm{op}} \to \mathbf{Cat}\right) \in \overline{\underline{\times}}\mathcal{L} \times \overline{\underline{\times}}\mathcal{L}$,

$$\overline{\Sigma}\left(\mathcal{L}' \underline{\times} \mathcal{L}''\right) \coloneqq \overline{\Sigma}\left(\mathcal{L}'\right) \times \overline{\Sigma}\left(\mathcal{L}''\right).$$

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

$$\overline{\Sigma} \circ \overline{\partial} = \mathrm{id}_{\mathrm{obj}\left(\mu\nu\mathsf{Poly}_{\Sigma_\mathcal{C}\mathcal{L}}\right)} \qquad \text{and} \qquad \overline{\partial} \circ \overline{\Sigma} = \mathrm{id}_{\overline{\underline{\times}}\mathcal{L}}.$$

∎

Lemma 3.17. *Let $\mathcal{L} : \mathcal{C}^{\mathrm{op}} \to \mathbf{Cat}$ be a strictly indexed category. The objects of $\mu\nu\mathsf{Poly}_{\Sigma_\mathcal{C}\mathcal{L}}$ with the functors that are induced by $\mu\nu\mathcal{L}$-indexed functors between objects of $\overline{\underline{\times}}\mathcal{L}$ form a subcategory of* $\mathbf{Cat}$.

Proof: Let $\mathcal{A}$ be an object of $\mu\nu\mathsf{Poly}_{\Sigma_\mathcal{C}\mathcal{L}}$. By Lemma 3.16, we have the associated strictly indexed category

$$\overline{\partial}\left(\mathcal{A}\right) = \mathcal{L}' : \mathcal{D}^{\mathrm{op}} \to \mathbf{Cat}.$$

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

$$\left(\mathrm{id}_\mathcal{D} : \mathcal{D} \to \mathcal{D}, \mathrm{id}\right) : \mathcal{L}' \to \mathcal{L}'$$

which is a $\mu\nu\mathcal{L}$-indexed category, since $\mathcal{L}'$ is a $\mu\nu\mathcal{L}$-indexed category by Lemma 3.15.

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

$$\left(\overline{E}, e\right) : \mathcal{L}' \to \mathcal{L}'' \qquad \text{and} \qquad \left(\overline{H}, h\right) : \mathcal{L}'' \to \mathcal{L}''',$$

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

$$\left(\overline{H} \circ \overline{E}, h_{\overline{E}^{\mathrm{op}}} \circ e\right)$$

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

Definition 3.18. We denote by $\overline{\mu\nu\mathsf{Poly}_{\Sigma_{\mathcal{C}}\mathcal{L}}}$ the category defined in Lemma 3.17.

Theorem 3.9. *Let $\mathcal{L} : \mathcal{C}^{\mathrm{op}} \to \mathbf{Cat}$ be a $*$-indexed category. The category $\Sigma_{\mathcal{C}}\mathcal{L}$ has $\mu\nu$-polynomials.*

Proof: By Theorem 3.8, since $\mathcal{L}$ is a $*$-indexed category, any endomorphism $E : \Sigma_{\mathcal{C}}\mathcal{L} \to \Sigma_{\mathcal{C}}\mathcal{L}$ of the subcategory $\overline{\mu\nu\mathsf{Poly}_{\Sigma_{\mathcal{C}}\mathcal{L}}}$ has an initial algebra and a terminal coalgebra. Therefore, in order to complete the proof, it is enough to show that the morphisms of $\overline{\mu\nu\mathsf{Poly}_{\Sigma_{\mathcal{C}}\mathcal{L}}}$ satisfy the inductive properties of Definition 1.1.

Let $\mathcal{A}$, $\mathcal{A}'$ and $\mathcal{A}''$ be objects of $\mu\nu\mathsf{Poly}_{\Sigma_{\mathcal{C}}\mathcal{L}}$. By Lemma 3.16, we have the associated strictly indexed categories

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

Recall that $\mathcal{L}', \mathcal{L}''$ and $\mathcal{L}'''$ are $\mu\nu\mathcal{L}$-indexed categories by Lemma 3.15.

1. The unique functor $\mathcal{A} \to \mathbb{1}$ is induced by the unique indexed functor
$$\big(\mathcal{D} \to \mathbb{1},\, (\mathcal{L}'\,(W) \to \mathbb{1})_{W \in \mathcal{D}}\big)$$
between $\mathcal{L}$ and the terminal indexed category $\mathbb{1} : \mathbb{1} \to \mathbf{Cat}$. Since $\mathcal{D} \to \mathbb{1}$ is a morphism of $\mu\nu\mathsf{Poly}_{\mathcal{C}}$ and (for any $W \in \mathcal{D}$) $\mathcal{L}'\,(W) \to \mathbb{1}$ is a morphism of $\mu\nu\mathsf{Poly}_{\mathcal{L}}$, we have that the unique indexed functor is a $\mu\nu\mathcal{L}$-indexed functor.
2. Given a functor $F : \mathbb{1} \to \mathcal{A} \cong \Sigma_{\mathcal{C}}\mathcal{L}'$, it corresponds to an object $(W \in \mathcal{D}, x \in \mathcal{L}'(W)) \in \Sigma_{\mathcal{C}}\mathcal{L}'$. In other words, F is induced by the strictly indexed functor
$$(W : \mathbb{1} \to \mathcal{D}, w : \mathbb{1} \to \mathcal{L}'(W))$$
in which W and w denote the obvious functors. Since any functor
$$\mathbb{1} \to \mathcal{D}$$
is a morphism of $\mu\nu\mathsf{Poly}_{\mathcal{C}}$ and (for any $W \in \mathcal{D}$) any functor
$$\mathbb{1} \to \mathcal{L}'(W)$$
is a morphism of $\mu\nu\mathsf{Poly}_{\mathcal{L}}$, we have that
$$(W : \mathbb{1} \to \mathcal{D}, w : \mathbb{1} \to \mathcal{L}'(W))$$

is a $\mu\nu\mathcal{L}$-indexed functor.

3. By Proposition 3.1, the binary product $\times : \Sigma_{\mathcal{C}}\mathcal{L} \times \Sigma_{\mathcal{C}}\mathcal{L} \to \Sigma_{\mathcal{C}}\mathcal{L}$ is induced by the strictly indexed functor

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

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

$$\begin{tikzcd} \mathcal{L}(W) \times \mathcal{L}(W') \arrow[r, "\mathcal{L}(\pi_1) \times \mathcal{L}(\pi_2)"] \arrow[dr, "p_{(W,W')}"'] & \mathcal{L}(W \times W') \times \mathcal{L}(W \times W') \arrow[d, "+"] \\ & \mathcal{L}(W \times W') \end{tikzcd}$$

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

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

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

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

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

is a morphism of $\mu\nu\mathsf{Poly}_{\mathcal{L}}$. Thus, since

$$\times : \mathcal{L}(W \times W') \times \mathcal{L}(W \times W') \to \mathcal{L}(W \times W')$$

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

4. By Corollary 3.6, the coproduct $\sqcup : \Sigma_{\mathcal{C}}\mathcal{L} \times \Sigma_{\mathcal{C}}\mathcal{L} \to \Sigma_{\mathcal{C}}\mathcal{L}$ is induced by the strictly indexed functor

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

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

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

of the extensive structure (see (3.4)) is a morphism of $\mu\nu\mathsf{Poly}_{\mathcal{L}}$.

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

5. The projections

$$\pi_1 : \mathcal{A} \times \mathcal{A}' \to \mathcal{A}, \qquad \pi_2 : \mathcal{A} \times \mathcal{A}' \to \mathcal{D}'$$

are, respectively, induced by the strictly indexed functors

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

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

$$\pi_1 : \mathcal{D} \times \mathcal{E} \to \mathcal{D}, \qquad \pi_2 : \mathcal{D} \times \mathcal{E} \to \mathcal{E}$$

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

$$\pi_1 : \mathcal{L}(W) \times \mathcal{L}(W') \to \mathcal{L}(W), \qquad \pi_2 : \mathcal{L}(W) \times \mathcal{L}(W') \to \mathcal{L}(W')$$

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

6. Assuming that $E : \mathcal{A} \to \mathcal{A}'$ and $J : \mathcal{A} \to \mathcal{A}''$ are functors induced by the $\mu\nu\mathcal{L}$-indexed functors

$$\left(\overline{E}, e : \mathcal{L}' \to \mathcal{L}'' \circ \overline{E}^{\mathrm{op}}\right) : \mathcal{L}' \to \mathcal{L}'' \qquad \text{and}$$
$$\left(\overline{J}, j : \mathcal{L}' \to \mathcal{L}''' \circ \overline{J}^{\mathrm{op}}\right) : \mathcal{L}' \to \mathcal{L}''',$$

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

$$\left(\left(\overline{E}, \overline{J}\right), (e, j)\right) : \mathcal{L}' \to \mathcal{L}'' \underline{\times} \mathcal{L}'''.$$

which is a $\mu\nu\mathcal{L}$-indexed functor as well since:
   - $\overline{E}$, $\overline{J}$ are morphisms of $\mu\nu\mathsf{Poly}_{\mathcal{C}}$ and, hence, so is $\left(\overline{E}, \overline{J}\right)$;
   - e_W, j_W are morphisms of $\mu\nu\mathsf{Poly}_{\mathcal{L}}$ for any $W \in \mathcal{D}$ and, hence, so is (e_W, j_W).

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

$$\left(\overline{H}, h\right) : \mathcal{L}' \underline{\times} \mathcal{L} \to \mathcal{L},$$

we have, by Theorem 3.8, that

8. μH is induced by the $\mu\nu\mathcal{L}$-indexed functor

$$\left(\mu\overline{H} : \mathcal{E} \to \mathcal{D},\, \mu\left(\underline{h}_{(-)}\right)\right) : \mathcal{L}' \to \mathcal{L}.$$

9. νH is induced by the $\mu\nu\mathcal{L}$-indexed functor

$$\left(\nu\overline{H} : \mathcal{E} \to \mathcal{D},\, \nu\left(\overline{h}_{(-)}\right)\right) : \mathcal{L}' \to \mathcal{L}.$$

■

Codually, we have:

Theorem 3.10. *Let $\mathcal{L} : \mathcal{C}^{\mathrm{op}} \to \mathbf{Cat}$ be a $*$-indexed category. The category $\Sigma_{\mathcal{C}}\mathcal{L}^{\mathrm{op}}$ has $\mu\nu$-polynomials.*

3.9. Σ-bimodel for function types, inductive and coinductive types. By Theorem 3.4, 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 3.4. More precisely:

Corollary 3.19. *Let $\mathcal{L} : \mathcal{C}^{\mathrm{op}} \to \mathbf{Cat}$ be a Σ-bimodel for inductive and coinductive types. The categories $\Sigma_{\mathcal{C}}\mathcal{L}$ and $\Sigma_{\mathcal{C}}\mathcal{L}^{\mathrm{op}}$ are distributive categories with $\mu\nu$-polynomials.*

Corollary 3.20. *Let $\mathcal{L} : \mathcal{C}^{\mathrm{op}} \to \mathbf{Cat}$ be a Σ-bimodel for function types, inductive and coinductive types. The categories $\Sigma_{\mathcal{C}}\mathcal{L}$ and $\Sigma_{\mathcal{C}}\mathcal{L}^{\mathrm{op}}$ are closed categories with $\mu\nu$-polynomials.*

4. Linear λ-calculus as an idealised AD target language

We describe a target language for our AD code transformations, a variant of the dependently typed enriched effect calculus [38, Chapter 5]. Its Cartesian types, linear types, and terms are generated by the grammar of Fig. 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 judgements $\Delta \mid \Gamma \vdash \tau : \mathrm{type}$ and $\Delta \mid \Gamma \vdash \underline{\tau} : \mathrm{ltype}$ for Cartesian and linear types, where $\Delta = \alpha_1 : \mathrm{type}, \ldots, \alpha_n : \mathrm{type}$ is a list of (Cartesian) type identifiers and $\Gamma = x_1 : \tau_1, \ldots, x_n : \tau_n$ is a list of identifiers x_i with Cartesian type τ_i. These kinding judgements are defined according to the rules displayed in Fig. 2 and 7.

We use typing judgements $\Delta \mid \Gamma \vdash t : \tau$ and $\Delta \mid \Gamma; \mathsf{v} : \underline{\tau} \vdash s : \underline{\sigma}$ for terms of well-kinded Cartesian types $\Delta \mid \Gamma \vdash \tau : \mathrm{type}$ and linear type $\Delta \mid \Gamma \vdash \underline{\sigma} : \mathrm{ltype}$, where $\Delta = \alpha_1 : \mathrm{type}, \ldots, \alpha_n : \mathrm{type}$ is a list of Cartesian type identifiers, $\Gamma = x_1 : \tau_1, \ldots, x_n : \tau_n$ is a list of identifiers x_i of well-kinded Cartesian type $\Delta \mid x_1 : \tau_1, \ldots, x_{i-1} : \tau_{i-1} \vdash \tau_i : \mathrm{type}$ and v is the unique linear identifier of well-kinded linear type $\Delta \mid \Gamma \vdash \underline{\tau} : \mathrm{ltype}$. Note that terms of linear type

$\kappa, \kappa', \kappa''$	$::=$	kinds
	type	kind of Cartesian types
	$\mid$ ltype	kind of linear types
$\underline{\tau}, \underline{\sigma}, \underline{\rho}$	$::=$	linear types
	$\underline{\alpha}$	linear type variable
	$\mid\ \underline{\mathbf{real}}^n$	real array
	$\mid\ \underline{\mathbf{1}}$	unit type
	$\mid\ \underline{\tau} * \underline{\sigma}$	binary product
	$\mid\ \Pi x : \tau . \underline{\sigma}$	power
	$\mid\ \Sigma x : \tau . \underline{\sigma}$	copower
	$\mid\ \mathbf{case}\, t\, \mathbf{of}\, \{\ell_1 x_1 \to \underline{\tau}_1 \mid \cdots \mid \ell_n x_n \to \underline{\tau}_n\}$	case distinction
	$\mid\ \underline{\mu}\underline{\alpha}.\underline{\tau}$	inductive type
	$\mid\ \underline{\nu}\underline{\alpha}.\underline{\tau}$	coinductive type
τ, σ, ρ	$::=$	Cartesian types
	$\ldots$	as in Fig. 1
	$\mid\ \underline{\tau} \multimap \underline{\sigma}$	linear function
	$\mid\ \Pi x : \tau . \sigma$	dependent function
	$\mid\ \Sigma x : \tau . \sigma$	dependent pair
t, s, r	$::=$ terms	
	$\ldots$	as in Fig. 1
	$\mid\ \mathsf{v}$	linear identifier
	$\mid\ \mathbf{let}\, \mathsf{v} =\, t\, \mathbf{in}\, s$	linear let-binding
	$\mid\ \mathsf{lop}(t_1, \ldots, t_k; s)$	linear operation
	$\mid\ !t \otimes s \mid \mathbf{case}\, t\, \mathbf{of}\, !y \otimes \mathsf{v} \to s$	copower intro/elim
	$\mid\ \underline{\lambda}\mathsf{v}.t \mid t \bullet s$	abstraction/application
	$\mid\ \underline{0} \mid t + s$	monoid structure
	$\mid\ \mathbf{bunch}\, (t, s)\, \mathbf{of}\, \{(\ell_1 x_1, \mathsf{v}) \to r_1 \mid \cdots \mid (\ell_n x_n, \mathsf{v}) \to r_n\}$	extensive families

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

always contain the unique linear identifier v in the typing context. These typing judgements are defined according to the rules displayed in Fig. 3, 8 and 9.

We work with linear operations $\mathsf{lop} \in \mathsf{LOp}^{m_1,\ldots,m_r}_{n_1,\ldots,n_k;n'_1,\ldots,n'_l}$, 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 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 $\mathrm{op} \in \mathsf{Op}^{m_1,\ldots,m_r}_{n_1,\ldots,n_k}$,

$$D\mathrm{op} \in \mathsf{LOp}^{m}_{n_1,\ldots,n_k;n_1,\ldots,n_k} \qquad\qquad D\mathrm{op}^t \in \mathsf{LOp}^{n_1,\ldots,n_k}_{n_1,\ldots,n_k;m}.$$

$$\frac{\Delta \vdash \tau : \mathrm{type}}{\Delta \mid \Gamma \vdash \tau : \mathrm{type}} \qquad \frac{((\alpha : \mathrm{type}) \in \Delta)}{\Delta \mid \Gamma \vdash \underline{\alpha} : \mathrm{ltype}}$$

$$\frac{}{\Delta \mid \Gamma \vdash \underline{\mathbf{real}}^n : \mathrm{ltype}} \qquad \frac{}{\Delta \mid \Gamma \vdash \underline{\mathbf{1}} : \mathrm{ltype}} \qquad \frac{\Delta \mid \Gamma \vdash \underline{\tau} : \mathrm{ltype} \quad \Delta \mid \Gamma \vdash \underline{\sigma} : \mathrm{ltype}}{\Delta \mid \Gamma \vdash \underline{\tau} * \underline{\sigma} : \mathrm{ltype}}$$

$$\frac{\cdot \mid \Gamma \vdash \tau : \mathrm{type} \quad \cdot \mid \Gamma, x : \tau \vdash \underline{\sigma} : \mathrm{ltype}}{\cdot \mid \Gamma \vdash \Pi x : \tau . \underline{\sigma} : \mathrm{ltype}} \qquad \frac{\cdot \mid \Gamma \vdash \tau : \mathrm{type} \quad \cdot \mid \Gamma, x : \tau \vdash \underline{\sigma} : \mathrm{ltype}}{\cdot \mid \Gamma \vdash \Sigma x : \tau . \underline{\sigma} : \mathrm{ltype}}$$

$$\frac{\Delta \mid \Gamma \vdash t : \{\ell_1 \tau_1 \mid \cdots \mid \ell_n \tau_n\} \quad \{\Delta \mid \Gamma, x_i : \tau_i \vdash \underline{\sigma}_i : \mathrm{ltype}\}_{1 \le i \le n}}{\Delta \mid \Gamma \vdash \mathbf{case}\, t\, \mathbf{of}\, \{\ell_1 x_1 \to \underline{\sigma}_1 \mid \cdots \mid \ell_n x_n \to \underline{\sigma}_n\} : \mathrm{ltype}}$$

$$\frac{\Delta, \alpha : \mathrm{type} \mid \Gamma \vdash \underline{\tau} : \mathrm{ltype}}{\Delta \mid \Gamma \vdash \underline{\mu \alpha . \tau} : \mathrm{ltype}} \qquad \frac{\Delta, \alpha : \mathrm{type} \mid \Gamma \vdash \underline{\tau} : \mathrm{ltype}}{\Delta \mid \Gamma \vdash \underline{\nu \alpha . \tau} : \mathrm{ltype}}$$

$$\frac{\cdot \mid \Gamma \vdash \underline{\tau} : \mathrm{ltype} \quad \cdot \mid \Gamma \vdash \underline{\sigma} : \mathrm{ltype}}{\cdot \mid \Gamma \vdash \underline{\tau} \multimap \underline{\sigma} : \mathrm{type}}$$

$$\frac{\cdot \mid \Gamma \vdash \tau : \mathrm{type} \quad \cdot \mid \Gamma, x : \tau \vdash \sigma : \mathrm{type}}{\cdot \mid \Gamma \vdash \Pi x : \tau . \sigma : \mathrm{type}} \qquad \frac{\cdot \mid \Gamma \vdash \tau : \mathrm{type} \quad \cdot \mid \Gamma, x : \tau \vdash \sigma : \mathrm{type}}{\cdot \mid \Gamma \vdash \Sigma x : \tau . \sigma : \mathrm{type}}$$

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 judgements 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 Γ 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 non-parameterized types (shaded in grey).

We will use these linear operations $D\mathrm{op}$ and $D\mathrm{op}^t$ as the forward and reverse derivatives of the corresponding primitive operations op*. We write

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

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

Fig. 5 and 11 display the equational theory we consider for the terms and types, which we call $(\alpha)\beta\eta+$-equivalence. To present this equational theory,

*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 [40, 39]. 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.

$$\frac{}{\Delta \mid \Gamma;\mathsf{v}:\underline{\tau}\vdash \mathsf{v}:\underline{\tau}} \qquad \frac{\Delta\mid\Gamma\vdash t:\underline{\tau}\quad \Delta\mid\Gamma,\mathsf{v}:\underline{\tau}\vdash s:\underline{\sigma}}{\Delta\mid\Gamma\vdash \mathbf{let}\,\mathsf{v}=t\,\mathbf{in}\,s:\underline{\sigma}} \qquad \frac{\Delta\mid\Gamma\vdash t:\tau\quad \Delta\mid\Gamma,x:\tau;\mathsf{v}:\underline{\sigma}\vdash s:\underline{\rho}}{\Delta\mid\Gamma;\mathsf{v}:\underline{\sigma}\vdash \mathbf{let}\,x=t\,\mathbf{in}\,s:\underline{\rho}}$$

$$\frac{\{\Delta\mid\Gamma\vdash t_i:\mathbf{real}^{n_i}\}_{i=1}^{k}\quad \Delta\mid\Gamma;\mathsf{v}:\underline{\tau}\vdash s:\mathbf{LDom}(\mathsf{lop})\quad (\mathsf{lop}\in\mathsf{LOp}^{m_1,\ldots,m_r}_{n_1,\ldots,n_k;n'_1,\ldots,n'_l})}{\Delta\mid\Gamma;\mathsf{v}:\underline{\tau}\vdash\mathsf{lop}(t_1,\ldots,t_k;s):\mathbf{CDom}(\mathsf{lop})}$$

$$\frac{}{\Delta\mid\Gamma;\mathsf{v}:\underline{\tau}\vdash\langle\rangle:\underline{\mathbf{1}}} \qquad \frac{\Delta\mid\Gamma;\mathsf{v}:\underline{\tau}\vdash t:\underline{\sigma}\quad \Delta\mid\Gamma;\mathsf{v}:\underline{\tau}\vdash s:\underline{\rho}}{\Delta\mid\Gamma;\mathsf{v}:\underline{\tau}\vdash\langle t,s\rangle:\underline{\sigma}*\underline{\rho}} \qquad \frac{\Delta\mid\Gamma;\mathsf{v}:\underline{\tau}\vdash t:\underline{\sigma}*\underline{\rho}}{\Delta\mid\Gamma;\mathsf{v}:\underline{\tau}\vdash\mathbf{fst}\,t:\underline{\sigma}}$$

$$\frac{\Delta\mid\Gamma;\mathsf{v}:\underline{\tau}\vdash t:\underline{\sigma}*\underline{\rho}}{\Delta\mid\Gamma;\mathsf{v}:\underline{\tau}\vdash\mathbf{snd}\,t:\underline{\rho}} \qquad \frac{\Delta\mid\Gamma,y:\sigma;\mathsf{v}:\underline{\tau}\vdash t:\underline{\rho}}{\Delta\mid\Gamma;\mathsf{v}:\underline{\tau}\vdash\lambda y.t:\Pi y:\sigma.\underline{\rho}}$$

$$\frac{\Delta\mid\Gamma;\mathsf{v}:\underline{\tau}\vdash t:\Pi y:\sigma.\underline{\rho}\quad\Delta\mid\Gamma\vdash s:\sigma}{\Delta\mid\Gamma;\mathsf{v}:\underline{\tau}\vdash t\,s:\underline{\rho}[^{s}/_{y}]} \qquad \frac{\Delta\mid\Gamma\vdash t:\sigma\quad\Delta\mid\Gamma;\mathsf{v}:\underline{\tau}\vdash s:\underline{\rho}[^{t}/_{y}]}{\Delta\mid\Gamma;\mathsf{v}:\underline{\tau}\vdash !t\otimes s:\Sigma y:\sigma.\underline{\rho}}$$

$$\frac{\Delta\mid\Gamma;\mathsf{v}:\underline{\tau}\vdash t:\Sigma y:\sigma.\underline{\rho}\quad\Delta\mid\Gamma,y:\sigma;\mathsf{v}:\underline{\rho}\vdash s:\underline{\rho}'\quad\Delta\mid\Gamma\mid\cdot\vdash\underline{\rho}':\mathrm{ltype}}{\Delta\mid\Gamma;\mathsf{v}:\underline{\tau}\vdash\mathbf{case}\,t\,\mathbf{of}\,!y\otimes\mathsf{v}\to s:\underline{\rho}'} \qquad \frac{\Delta\mid\Gamma;\mathsf{v}:\underline{\tau}\vdash t:\underline{\sigma}}{\Delta\mid\Gamma\vdash\underline{\lambda}\mathsf{v}.t:\underline{\tau}\multimap\underline{\sigma}}$$

$$\frac{\Delta\mid\Gamma\vdash t:\underline{\rho}\multimap\underline{\sigma}\quad\Delta\mid\Gamma;\mathsf{v}:\underline{\tau}\vdash s:\underline{\rho}}{\Delta\mid\Gamma;\mathsf{v}:\underline{\tau}\vdash t\bullet s:\underline{\sigma}} \qquad \frac{}{\Delta\mid\Gamma;\mathsf{v}:\underline{\tau}\vdash\underline{0}:\underline{\sigma}} \qquad \frac{\Delta\mid\Gamma;\mathsf{v}:\underline{\tau}\vdash t:\underline{\sigma}\quad\Delta\mid\Gamma;\mathsf{v}:\underline{\tau}\vdash s:\underline{\sigma}}{\Delta\mid\Gamma;\mathsf{v}:\underline{\tau}\vdash t+s:\underline{\sigma}}$$

$$\frac{\begin{array}{l}\Delta\mid\Gamma\vdash t:\{\ell_1\tau_1\mid\cdots\mid\ell_n\tau_n\}\\ \Delta\mid\Gamma;\mathsf{v}:\underline{\sigma}\vdash s:\mathbf{case}\,t\,\mathbf{of}\,\{\ell_1x_1\to\underline{\sigma}_1\mid\cdots\mid\ell_nx_n\to\underline{\sigma}_n\}\\ \left\{\Delta\mid\Gamma,x_i:\tau_i;\mathsf{v}:\underline{\sigma}_i\vdash r_i:\underline{\rho}_i\right\}_{1\le i\le n}\end{array}}{\begin{array}{r}\Delta\mid\Gamma;\mathsf{v}:\underline{\sigma}\vdash\mathbf{bunch}\,(t,s)\,\mathbf{of}\,\{(\ell_1x_1,\mathsf{v})\to r_1\mid\cdots\mid(\ell_nx_n,\mathsf{v})\to r_n\}\\ :\mathbf{case}\,t\,\mathbf{of}\,\{\ell_1x_1\to\underline{\rho}_1\mid\cdots\mid\ell_nx_n\to\underline{\rho}_n\}\end{array}}$$

$$\frac{\Delta\mid\Gamma;\mathsf{v}:\underline{\rho}\vdash t:\underline{\tau}[^{\underline{\mu}\alpha.\underline{\tau}}/_{\alpha}]}{\Delta\mid\Gamma;\mathsf{v}:\underline{\rho}\vdash\mathbf{roll}\,t:\underline{\mu}\alpha.\underline{\tau}} \qquad \frac{\Delta\mid\Gamma;\mathsf{v}:\underline{\rho}\vdash t:\underline{\mu\alpha}.\underline{\tau}\quad\Delta\mid\Gamma;\mathsf{v}:\underline{\tau}[^{\underline{\sigma}}/_{\alpha}]\vdash s:\underline{\sigma}}{\Delta\mid\Gamma;\mathsf{v}:\underline{\rho}\vdash\mathbf{fold}\,t\,\mathbf{with}\,\mathsf{v}\to s:\underline{\sigma}}$$

$$\frac{\Delta\mid\Gamma;\mathsf{v}:\underline{\rho}\vdash t:\underline{\sigma}\quad\Delta\mid\Gamma;\mathsf{v}:\underline{\sigma}\vdash s:\underline{\tau}[^{\underline{\sigma}}/_{\alpha}]}{\Delta\mid\Gamma;\mathsf{v}:\underline{\rho}\vdash\mathbf{gen}\,\mathbf{from}\,t\,\mathbf{with}\,\mathsf{v}\to s:\underline{\nu\alpha}.\underline{\tau}} \qquad \frac{\Delta\mid\Gamma;\mathsf{v}:\underline{\rho}\vdash t:\underline{\nu\alpha}.\underline{\tau}}{\Delta\mid\Gamma;\mathsf{v}:\underline{\rho}\vdash\mathbf{unroll}\,t:\underline{\tau}[^{\underline{\nu\alpha}.\underline{\tau}}/_{\alpha}]}$$

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

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

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

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

we define in Fig. 10, by induction, some syntactic sugar for the functorial action $\Delta, \Delta' \mid \Gamma; \mathsf{v} : \underline{\tau}[{}^{\underline{\sigma}}/_{\underline{\alpha}}] \vdash \underline{\tau}[{}^{\mathsf{v}\vdash t}/_{\underline{\alpha}}] : \underline{\tau}[{}^{\underline{\rho}}/_{\underline{\alpha}}]$ in argument $\underline{\alpha}$ of parameterized types $\Delta, \underline{\alpha} : \mathrm{type} \vdash \underline{\tau} : \mathrm{ltype}$ on terms $\Delta' \mid \Gamma; \mathsf{v} : \underline{\sigma} \vdash t : \underline{\rho}$.

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

FIGURE 10. Functorial action $\Delta, \Delta' \mid \Gamma; \mathsf{v} : \underline{\tau}[{}^{\underline{\sigma}}/_{\underline{\alpha}}] \vdash \underline{\tau}[{}^{\mathsf{v}\vdash t}/_{\underline{\alpha}}] : \underline{\tau}[{}^{\underline{\rho}}/_{\underline{\alpha}}]$ in argument $\underline{\alpha}$ of parameterized types $\Delta, \underline{\alpha} : \mathrm{type} \vdash \underline{\tau} : \mathrm{type}$ on terms $\Delta' \mid \Gamma; \mathsf{v} : \underline{\sigma} \vdash t : \underline{\rho}$ of the target language.

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

- $\mathbf{CSyn}$ 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 $\cdot \mid p : \tau \vdash \underline{\sigma} : \mathrm{ltype}$ up to $(\alpha)\beta\eta+$-equivalence.
- Morphisms in $\mathbf{LSyn}(\tau)(\underline{\sigma}, \underline{\rho})$ are terms $\cdot \mid x : \tau; \mathsf{v} : \underline{\sigma} \vdash t : \underline{\rho}$ modulo $(\alpha)\beta\eta+$-equivalence.
- Identities in $\mathbf{LSyn}(\tau)$ are represented by the terms $\cdot \mid x : \tau; \mathsf{v} : \underline{\sigma} \vdash \mathsf{v} : \underline{\sigma}$.
- Composition of $\cdot \mid x : \tau; \mathsf{v} : \underline{\sigma}_1 \vdash t : \underline{\sigma}_2$ and $\cdot \mid x : \tau; \mathsf{v} : \underline{\sigma}_2 \vdash s : \underline{\sigma}_3$ in $\mathbf{LSyn}(\tau)$ is defined as $\cdot \mid x : \tau; \mathsf{v} : \underline{\sigma}_1 \vdash \mathbf{let}\,\mathsf{v} = t\,\mathbf{in}\,s : \underline{\sigma}_3$.
- Change of base $\mathbf{LSyn}(t) : \mathbf{LSyn}(\tau) \to \mathbf{LSyn}(\tau')$ along $(\cdot \mid x' : \tau' \vdash t : \tau) \in \mathbf{CSyn}(\tau', \tau)$ is defined $\mathbf{LSyn}(t)(\cdot \mid x : \tau; \mathsf{v} : \underline{\sigma} \vdash s : \underline{\rho}) \stackrel{\mathrm{def}}{=} \cdot \mid x' : \tau'; \mathsf{v} : \underline{\sigma} \vdash \mathbf{let}\,x = t\,\mathbf{in}\,s \; : \; \underline{\rho}$.

$$\mathbf{let}\,\mathsf{v} = t\,\mathbf{in}\,s = s[{}^{t}/_{\mathsf{v}}]$$

$$\mathbf{case}\,!t \otimes s\,\mathbf{of}\,!x \otimes \mathsf{v} \to r = r[{}^{t}/_{x}, {}^{s}/_{\mathsf{v}}] \qquad t[{}^{s}/_{\mathsf{v}}] \overset{\#y}{=} \mathbf{case}\,s\,\mathbf{of}\,!y \otimes \mathsf{v} \to t[{}^{!y\otimes\mathsf{v}}/_{\mathsf{v}}]$$

$$(\underline{\lambda}\mathsf{v}.t) \bullet s = t[{}^{s}/_{\mathsf{v}}] \qquad t = \underline{\lambda}\mathsf{v}.t \bullet \mathsf{v}$$

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

$$(\Gamma; \mathsf{v} : \underline{\tau} \vdash t : \underline{\sigma}) \Rightarrow t[{}^{\underline{0}}/_{\mathsf{v}}] = \underline{0} \qquad (\Gamma; \mathsf{v} : \underline{\tau} \vdash t : \underline{\sigma}) \Rightarrow t[{}^{s+r}/_{\mathsf{v}}] = t[{}^{s}/_{\mathsf{v}}] + t[{}^{r}/_{\mathsf{v}}]$$

$$\mathbf{case}\,\ell_i t\,\mathbf{of}\,\{\ell_1 x_1 \to \underline{\tau}_1 \mid \cdots \mid \ell_n x_n \to \underline{\tau}_n\} = \underline{\tau}_i[{}^{t}/_{x_i}]$$

$$\underline{\tau}[{}^{t}/_{y}] \overset{\#x_1,\ldots,x_n}{=} \mathbf{case}\,t\,\mathbf{of}\,\{\ell_1\,x_1 \to \underline{\tau}[{}^{\ell_1\,x_1}/_{y}] \mid \cdots \mid \ell_{\mathsf{n}}\,x_n \to \underline{\tau}[{}^{\ell_{\mathsf{n}}\,x_n}/_{y}]\}$$

$$\mathbf{bunch}\,(\ell_i t, s)\,\mathbf{of}\,\{(\ell_1 x_1, \mathsf{v}) \to r_1 \mid \cdots \mid (\ell_n x_n, \mathsf{v}) \to r_n\} = r_i[{}^{t}/_{x_i}, {}^{s}/_{\mathsf{v}}]$$

$$r[{}^{t}/_{x}, {}^{s}/_{\mathsf{v}}] \overset{\#x_1,\ldots,x_n}{=} \mathbf{bunch}\,(t, s)\,\mathbf{of}\,\{(\ell_1 x_1, \mathsf{v}) \to r[{}^{\ell_1 x_1}/_{x}] \mid \cdots \mid (\ell_n x_n, \mathsf{v}) \to r[{}^{\ell_n x_n}/_{x}]\}$$

$$\mathbf{fold}\,\mathbf{roll}\,t\,\mathbf{with}\,\mathsf{v} \to s = s[{}^{\underline{\tau}[{}^{\mathsf{v}\vdash \mathbf{fold}\,\mathsf{v}\,\mathbf{with}\,\mathsf{v}\to s}/_{\underline{\alpha}}][{}^{t}/_{\mathsf{v}}]}/_{\mathsf{v}}]$$

$$r[{}^{\mathbf{roll}\,\mathsf{v}}/_{\mathsf{v}}] = s[{}^{\underline{\tau}[{}^{\mathsf{v}\vdash r}/_{\underline{\alpha}}]}/_{\mathsf{v}}] \Rightarrow r[{}^{t}/_{\mathsf{v}}] = \mathbf{fold}\,t\,\mathbf{with}\,\mathsf{v} \to s$$

$$\mathbf{unroll}\,(\mathbf{gen}\,\mathbf{from}\,t\,\mathbf{with}\,\mathsf{v} \to s) = \underline{\tau}[{}^{\mathsf{v}\vdash \mathbf{gen}\,\mathbf{from}\,\mathsf{v}\,\mathbf{with}\,\mathsf{v}\to s}/_{\underline{\alpha}}][{}^{s}/_{\mathsf{v}}, {}^{t}/_{\mathsf{v}}]$$

$$\mathbf{unroll}\,r = \underline{\tau}[{}^{\mathsf{v}\vdash r}/_{\underline{\alpha}}][{}^{s}/_{\mathsf{v}}] \Rightarrow r[{}^{t}/_{\mathsf{v}}] = \mathbf{gen}\,\mathbf{from}\,t\,\mathbf{with}\,\mathsf{v} \to s$$

FIGURE 11. Equational rules for the idealised, linear AD language, which we use on top of the rules of Fig. 5. In addition to standard $\beta\eta$-rules for $!(-) \otimes (-)$- and $\multimap$-types, we add rules making $(\underline{0}, +)$ 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.

- All type formers are interpreted as one expects based on their notation, using introduction and elimination rules for the required structural isomorphisms.

Corollary 4.1. *$\Sigma_{\mathbf{CSyn}}\mathbf{LSyn}$ and $\Sigma_{\mathbf{CSyn}}\mathbf{LSyn}^{op}$ are both bi-Cartesian closed categories with $\mu\nu$-polynomials.*

In fact, $\mathbf{LSyn} : \mathbf{CSyn}^{op} \to \mathbf{Cat}$ 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 that for any other such a

Σ-bimodel $\mathcal{L} : \mathcal{C}^{op} \to \mathbf{Cat}$, we have a unique homomorphism $([\![-]\!], [\![-]\!]) : (\mathbf{CSyn}, \mathbf{LSyn}) \to (\mathcal{C}, \mathcal{L})$.

Corollary 4.2 (Universal property of $(\mathbf{CSyn}, \mathbf{LSyn})$)**.** *For any Σ-bimodel $\mathcal{L} : \mathcal{C}^{op} \to \mathbf{Cat}$ of tuples, self-dual primitive types and primitive operations, function types, sum types and inductive and coinductive types, we obtain canonical bi-Cartesian closed functors that preserve $\mu\nu$-polynomials*

$$\Sigma_{[\![-]\!]}[\![-]\!] : \Sigma_{\mathbf{CSyn}}\mathbf{LSyn} \to \Sigma_{\mathcal{C}}\mathcal{L}, \qquad \Sigma_{[\![-]\!]}[\![-]\!]^{op} : \Sigma_{\mathbf{CSyn}}\mathbf{LSyn}^{op} \to \Sigma_{\mathcal{C}}\mathcal{L}^{op}.$$

5. Novel AD algorithms as source-code transformations

As $\Sigma_{\mathbf{CSyn}}\mathbf{LSyn}$ and $\Sigma_{\mathbf{CSyn}}\mathbf{LSyn}^{op}$ are both bi-Cartesian closed categories with $\mu\nu$-polynomials, by Corollary 4.1, the universal property of $\mathbf{Syn}$ (Corollary 2.1) yields unique structure-preserving functors $\overrightarrow{\mathcal{D}}(-) : \mathbf{Syn} \to \Sigma_{\mathbf{CSyn}}\mathbf{LSyn}$ and $\overleftarrow{\mathcal{D}}(-) : \mathbf{Syn} \to \Sigma_{\mathbf{CSyn}}\mathbf{LSyn}^{op}$ 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 5.1 (CHAD)**.** *Once we fix the derivatives of the ground types and primitive operations of* $\mathbf{Syn}$ *by defining*

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

we obtain unique functors

$$\overrightarrow{\mathcal{D}}(-) : \mathbf{Syn} \to \Sigma_{\mathbf{CSyn}}\mathbf{LSyn}, \qquad \overleftarrow{\mathcal{D}}(-) : \mathbf{Syn} \to \Sigma_{\mathbf{CSyn}}\mathbf{LSyn}^{op} \tag{5.1}$$

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

By definition of equality in $\mathbf{Syn}$, $\Sigma_{\mathbf{CSyn}}\mathbf{LSyn}$ and $\Sigma_{\mathbf{CSyn}}\mathbf{LSyn}^{op}$, *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.

5.1. Kinding and typing of the code transformations. We define for each type τ of the source language:

- a Cartesian type $\overrightarrow{\mathcal{D}}(\tau)_1$ 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 $\Gamma = x_1 : \tau_1, \ldots, x_n : \tau_n$ as

$$\begin{array}{ll}
\overrightarrow{\mathcal{D}}(\Gamma)_1 \stackrel{\text{def}}{=} x_1 : \overrightarrow{\mathcal{D}}(\tau_1)_1, \ldots, x_n : \overrightarrow{\mathcal{D}}(\tau_n)_n & \text{(a Cartesian typing context)} \\
\overrightarrow{\mathcal{D}}(\Gamma)_2 \stackrel{\text{def}}{=} \boldsymbol{(}\overrightarrow{\mathcal{D}}(\tau_1)_2[{}^{x_1}/_{p}] \ast \cdots \ast \overrightarrow{\mathcal{D}}(\tau_n)_2[{}^{x_n}/_{p}]\boldsymbol{)} & \text{(a linear type)} \\
\overleftarrow{\mathcal{D}}(\Gamma)_1 \stackrel{\text{def}}{=} x_1 : \overleftarrow{\mathcal{D}}(\tau_1)_1, \ldots, x_n : \overleftarrow{\mathcal{D}}(\tau_n)_n & \text{(a Cartesian typing context)} \\
\overleftarrow{\mathcal{D}}(\Gamma)_2 \stackrel{\text{def}}{=} \boldsymbol{(}\overleftarrow{\mathcal{D}}(\tau_1)_2[{}^{x_1}/_{p}] \ast \cdots \ast \overleftarrow{\mathcal{D}}(\tau_n)_2[{}^{x_n}/_{p}]\boldsymbol{)} & \text{(a linear type).}
\end{array}$$

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

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

Similarly, the functors $\overrightarrow{\mathcal{D}}(-) : \mathbf{Syn} \to \Sigma_{\mathbf{CSyn}}\mathbf{LSyn}$ and $\overleftarrow{\mathcal{D}}(-) : \mathbf{Syn} \to \Sigma_{\mathbf{CSyn}}\mathbf{LSyn}^{op}$ 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:

$$\begin{array}{l}\Delta \mid \overrightarrow{\mathcal{D}}(\Gamma)_1 \vdash \overrightarrow{\mathcal{D}}_{\overline{\Gamma}}(t)_1 : \overrightarrow{\mathcal{D}}(\tau)_1 \\ \Delta \mid \overrightarrow{\mathcal{D}}(\Gamma)_1; \mathsf{v} : \overrightarrow{\mathcal{D}}(\Gamma)_2 \vdash \overrightarrow{\mathcal{D}}_{\overline{\Gamma}}(t)_2 : \overrightarrow{\mathcal{D}}(\tau)_2[{}^{\overrightarrow{\mathcal{D}}_{\overline{\Gamma}}(t)_1}/_p] \\ \Delta \mid \overleftarrow{\mathcal{D}}(\Gamma)_1 \vdash \overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(t)_1 : \overleftarrow{\mathcal{D}}(\tau)_1 \\ \Delta \mid \overleftarrow{\mathcal{D}}(\Gamma)_1; \mathsf{v} : \overleftarrow{\mathcal{D}}(\tau)_2[{}^{\overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(t)_1}/_p] \vdash \overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(t)_2 : \overleftarrow{\mathcal{D}}(\Gamma)_2,\end{array}$$

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

However, as we noted already in Insight 1 of §, 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:

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

where $\overrightarrow{\mathcal{D}}_{\overline{\Gamma}}(t) \stackrel{\beta\eta+}{=} \langle \overrightarrow{\mathcal{D}}_{\overline{\Gamma}}(t)_1, \underline{\lambda}\mathsf{v}.\overrightarrow{\mathcal{D}}_{\overline{\Gamma}}(t)_2 \rangle$ and $\overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(t) \stackrel{\beta\eta+}{=} \langle \overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(t)_1, \underline{\lambda}\mathsf{v}.\overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(t)_2 \rangle$. 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 $\overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(t)$.

5.2. Some notation. In the rest of this section, we use the following syntactic sugar:

- a notation for (linear) n-ary tuple types:

$$(\underline{\tau}_1 * \ldots * \underline{\tau}_n) \stackrel{\text{def}}{=} (((\underline{\tau}_1 * \underline{\tau}_2) \cdots * \underline{\tau}_{n-1}) * \underline{\tau}_n);$$

- a notation for n-ary tuples: $\langle t_1, \cdots, t_n \rangle \stackrel{\text{def}}{=} \langle \langle \langle t_1, t_2 \rangle \cdots, t_{n-1} \rangle, t_n \rangle$;
- given $\Gamma; \mathsf{v} : \underline{\tau} \vdash t : (\underline{\sigma}_1 * \cdots * \underline{\sigma}_n)$, we write $\Gamma; \mathsf{v} : \underline{\tau} \vdash \mathbf{proj}_i\,(t) : \underline{\sigma}_i$ for the obvious i-th projection of t, which is constructed by repeatedly applying **fst** and **snd** to t;
- given $\Gamma; \mathsf{v} : \underline{\tau} \vdash t : \underline{\sigma}_i$, we write the i-th coprojection $\Gamma; \mathsf{v} : \underline{\tau} \vdash \mathbf{coproj}_i\,(t) \stackrel{\text{def}}{=} \langle \underline{0}, \ldots, \underline{0}, t, \underline{0}, \ldots, \underline{0} \rangle : (\underline{\sigma}_1 * \cdots * \underline{\sigma}_n)$;
- for a list $x_1, \ldots, x_n$ of distinct identifiers, we write $\mathbf{idx}(x_i; x_1, \ldots, x_n) \stackrel{\text{def}}{=} i$ 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.

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

5.3. Code transformations of primitive types and operations. We have suitable terms (linear operations)

$$
\begin{array}{llll}
x_1 : \mathbf{real}^{n_1}, \cdots, x_k : \mathbf{real}^{n_k} \;;\; \mathsf{v} : \underline{\mathbf{real}}^{n_1} * \cdots * \underline{\mathbf{real}}^{n_k} & \vdash D\mathrm{op}(x_1, \ldots, x_k; \mathsf{v}) & : \underline{\mathbf{real}}^m \\
x_1 : \mathbf{real}^{n_1}, \cdots, x_k : \mathbf{real}^{n_k} \;;\; \mathsf{v} : \underline{\mathbf{real}}^m & \vdash D\mathrm{op}^t(x_1, \ldots, x_k; \mathsf{v}) & : \underline{\mathbf{real}}^{n_1} * \cdots * \underline{\mathbf{real}}^{n_k}
\end{array}
$$

to represent the forward- and reverse-mode derivatives of the primitive operations $\mathrm{op} \in \mathsf{Op}^m_{n_1,\ldots,n_k}$. Using these, we define

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

$$
\begin{aligned}
\overrightarrow{\mathcal{D}}_{\overline{\Gamma}}(\mathrm{op}(t_1, \ldots, t_k)) \stackrel{\text{def}}{=}\; & \mathbf{let}\,\langle x_1, x_1'\rangle = \overrightarrow{\mathcal{D}}_{\overline{\Gamma}}(t_1)\,\mathbf{in} \cdots \mathbf{let}\,\langle x_k, x_k'\rangle = \overrightarrow{\mathcal{D}}_{\overline{\Gamma}}(t_k)\,\mathbf{in} \\
& \langle \mathrm{op}(x_1, \ldots, x_k), \underline{\lambda}\mathsf{v}.D\mathrm{op}(x_1, \ldots, x_n; \langle x_1' \bullet \mathsf{v}, \ldots, x_k' \bullet \mathsf{v}\rangle)\rangle
\end{aligned}
$$

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

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

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

$$
\begin{aligned}
[\![x_1, \ldots, x_k; y \vdash D\mathrm{op}(x_1, \ldots, x_k; \mathsf{v})]\!] &= D[\![\mathrm{op}]\!] \\
[\![x_1, \ldots, x_k; \mathsf{v} \vdash D\mathrm{op}^t(x_1, \ldots, x_k; \mathsf{v})]\!] &= D[\![\mathrm{op}]\!]^t.
\end{aligned}
$$

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

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

By Corollary 2.1, the extension of the AD transformations $\overrightarrow{\mathcal{D}}$ and $\overleftarrow{\mathcal{D}}$ to the full source language are now canonically determined, as the unique $\mu\nu$-polynimials preserving bi-Cartesian closed functors that extend the previous definitions.

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

$$\begin{aligned}
&\overrightarrow{\mathcal{D}}(\mathbf{1})_1 \stackrel{\text{def}}{=} \mathbf{1}\\
&\overrightarrow{\mathcal{D}}(\tau * \sigma)_1 \stackrel{\text{def}}{=} \overrightarrow{\mathcal{D}}(\tau)_1 * \overrightarrow{\mathcal{D}}(\sigma)_1\\
&\overrightarrow{\mathcal{D}}(\tau \to \sigma)_1 \stackrel{\text{def}}{=} \Pi p : \overrightarrow{\mathcal{D}}(\tau)_1 . \Sigma p' : \overrightarrow{\mathcal{D}}(\sigma)_1 . \overrightarrow{\mathcal{D}}(\tau)_2 \multimap \overrightarrow{\mathcal{D}}(\sigma)_2[{}^{p'}\!/_{p}]\\
&\overrightarrow{\mathcal{D}}(\{\ell_1\tau_1 \mid \cdots \mid \ell_n\tau_n\})_1 \stackrel{\text{def}}{=} \left\{\ell_1\overrightarrow{\mathcal{D}}(\tau_1)_1 \mid \cdots \mid \ell_n\overrightarrow{\mathcal{D}}(\tau_n)_1\right\}\\
&\overrightarrow{\mathcal{D}}(\alpha)_1 \stackrel{\text{def}}{=} \alpha\\
&\overrightarrow{\mathcal{D}}(\mu\alpha.\tau)_1 \stackrel{\text{def}}{=} \mu\alpha.\overrightarrow{\mathcal{D}}(\tau)_1\\
&\overrightarrow{\mathcal{D}}(\nu\alpha.\tau)_1 \stackrel{\text{def}}{=} \nu\alpha.\overrightarrow{\mathcal{D}}(\tau)_1
\end{aligned}$$

$$\begin{aligned}
&\overrightarrow{\mathcal{D}}(\mathbf{1})_2 \stackrel{\text{def}}{=} \underline{\mathbf{1}}\\
&\overrightarrow{\mathcal{D}}(\tau * \sigma)_2 \stackrel{\text{def}}{=} \overrightarrow{\mathcal{D}}(\tau)_2[{}^{\mathbf{fst}\,p}\!/_{p}] * \overrightarrow{\mathcal{D}}(\sigma)_2[{}^{\mathbf{snd}\,p}\!/_{p}]\\
&\overrightarrow{\mathcal{D}}(\tau \to \sigma)_2 \stackrel{\text{def}}{=} \Pi p' : \overrightarrow{\mathcal{D}}(\tau)_1 . \overrightarrow{\mathcal{D}}(\sigma)_2[{}^{\mathbf{fst}\,(p\,p')}\!/_{p}]\\
&\overrightarrow{\mathcal{D}}(\{\ell_1\tau_1 \mid \cdots \mid \ell_n\tau_n\})_2 \stackrel{\text{def}}{=} \mathbf{case}\, p\, \mathbf{of}\, \{\ell_1 p \to \overrightarrow{\mathcal{D}}(\tau_1)_2 \mid \cdots \mid \ell_n p \to \overrightarrow{\mathcal{D}}(\tau_n)_2\}\\
&\overrightarrow{\mathcal{D}}(\alpha)_2 \stackrel{\text{def}}{=} \underline{\alpha}\\
&\overrightarrow{\mathcal{D}}(\mu\alpha.\tau)_2 \stackrel{\text{def}}{=} \underline{\mu}\underline{\alpha}.\overrightarrow{\mathcal{D}}(\tau)_2[{}^{\mathbf{fold}\, p\, \mathbf{with}\, y \to \overrightarrow{\mathcal{D}}(\tau)_1[{}^{y \vdash \mathbf{roll}\, y}\!/_{\alpha}]}\!/_{p}]\\
&\overrightarrow{\mathcal{D}}(\nu\alpha.\tau)_2 \stackrel{\text{def}}{=} \underline{\nu}\underline{\alpha}.\overrightarrow{\mathcal{D}}(\tau)_2[{}^{\mathbf{unroll}\, p}\!/_{p}]
\end{aligned}$$

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

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

$$\overrightarrow{\mathcal{D}}_{\overline{\Gamma}}(\mathbf{let}\, x = t\, \mathbf{in}\, s) \stackrel{\text{def}}{=} \begin{array}{l}\mathbf{let}\, \langle x, x' \rangle = \overrightarrow{\mathcal{D}}_{\overline{\Gamma}}(t)\, \mathbf{in} \\ \mathbf{let}\, \langle y, y' \rangle = \overrightarrow{\mathcal{D}}_{\overline{\Gamma},x}(s)\, \mathbf{in} \\ \langle y, \underline{\lambda}\mathsf{v}.y' \bullet \langle \mathsf{v}, x' \bullet \mathsf{v}\rangle\rangle\end{array}$$

$$\overrightarrow{\mathcal{D}}_{\overline{\Gamma}}(\langle\rangle) \stackrel{\text{def}}{=} \langle\langle\rangle, \underline{\lambda}\mathsf{v}.\langle\rangle\rangle$$

$$\overrightarrow{\mathcal{D}}_{\overline{\Gamma}}(\langle t, s\rangle) \stackrel{\text{def}}{=} \begin{array}{l}\mathbf{let}\, \langle x, x' \rangle = \overrightarrow{\mathcal{D}}_{\overline{\Gamma}}(t)\, \mathbf{in} \\ \mathbf{let}\, \langle y, y' \rangle = \overrightarrow{\mathcal{D}}_{\overline{\Gamma}}(s)\, \mathbf{in} \\ \langle\langle x, y\rangle, \underline{\lambda}\mathsf{v}.\langle x' \bullet \mathsf{v}, y' \bullet \mathsf{v}\rangle\rangle\end{array}$$

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

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

$$\overrightarrow{\mathcal{D}}_{\overline{\Gamma}}(\lambda x.t) \stackrel{\text{def}}{=} \begin{array}{l}\mathbf{let}\, y = \lambda x.\overrightarrow{\mathcal{D}}_{\overline{\Gamma},x}(t)\, \mathbf{in} \\ \langle \lambda x.\mathbf{let}\, \langle z, z' \rangle = y\, x\, \mathbf{in}\, \langle z, \underline{\lambda}\mathsf{v}.z' \bullet \langle \underline{0}, \mathsf{v}\rangle\rangle, \underline{\lambda}\mathsf{v}.\lambda x.(\mathbf{snd}\, (y\, x)) \bullet \langle \mathsf{v}, \underline{0}\rangle\rangle\end{array}$$

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

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

$$\overrightarrow{\mathcal{D}}_{\overline{\Gamma}}(\mathbf{case}\, t\, \mathbf{of}\, \{\ell_1 x_1 \to s_1 \mid \cdots \mid \ell_n x_n \to s_n\}) \stackrel{\text{def}}{=} \begin{array}{l}\mathbf{let}\, \langle y, y' \rangle = \overrightarrow{\mathcal{D}}_{\overline{\Gamma}}(t)\, \mathbf{in} \\ \mathbf{case}\, y\, \mathbf{of}\, \{\ell_1 x_1 \to \\ \quad \mathbf{let}\, \langle z_1, z_1' \rangle = \overrightarrow{\mathcal{D}}_{\overline{\Gamma},x_1}(s_1)\, \mathbf{in} \\ \quad \langle z_1, \underline{\lambda}\mathsf{v}.\mathbf{bunch}\, (y, \langle \mathsf{v}, y' \bullet \mathsf{v}\rangle)\, \mathbf{of}\, \{ \\ \qquad\qquad (\ell_1 x_1, \mathsf{v}) \to z_1' \bullet \mathsf{v} \\ \qquad\qquad \mid (\ell_2 x_2, \mathsf{v}) \to \underline{0} \\ \qquad\qquad \mid \cdots \\ \qquad\qquad \mid (\ell_n x_n, \mathsf{v}) \to \underline{0}\}\rangle \\ \qquad \mid \cdots \mid \\ \qquad \ell_n x_n \to \\ \quad \mathbf{let}\, \langle z_n, z_n' \rangle = \overrightarrow{\mathcal{D}}_{\overline{\Gamma},x_1}(s_1)\, \mathbf{in} \\ \quad \langle z_n, \underline{\lambda}\mathsf{v}.\mathbf{bunch}\, (y, \langle \mathsf{v}, y' \bullet \mathsf{v}\rangle)\, \mathbf{of}\, \{ \\ \qquad\qquad (\ell_1 x_1, \mathsf{v}) \to \underline{0} \\ \qquad\qquad \mid \cdots \\ \qquad\qquad \mid (\ell_{n-1} x_{n-1}, \mathsf{v}) \to \underline{0} \\ \qquad\qquad \mid (\ell_n x_n, \mathsf{v}) \to z_n' \bullet \mathsf{v}\}\rangle\}\end{array}$$

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

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

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

$$\begin{aligned}\overrightarrow{\mathcal{D}}_{\overline{\Gamma}}(\mathbf{gen}\,\mathbf{from}\,t\,\mathbf{with}\,x \to s) \stackrel{\text{def}}{=}\ & \mathbf{let}\,\langle y, y'\rangle = \overrightarrow{\mathcal{D}}_{\overline{\Gamma}}(t)\,\mathbf{in} \\ & \mathbf{let}\,z = \lambda x.\overrightarrow{\mathcal{D}}_{x}(s)\,\mathbf{in} \\ & \langle \mathbf{gen}\,\mathbf{from}\,y\,\mathbf{with}\,x \to \mathbf{fst}\,(z\,x), \\ & \underline{\lambda}\mathsf{v}.\mathbf{gen}\,\mathbf{from}\,y' \bullet \mathsf{v}\,\mathbf{with}\,\mathsf{v} \to (\mathbf{snd}\,(z\,y)) \bullet \mathsf{v}\rangle\end{aligned}$$

5.5. 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:

$$\begin{aligned}
&\overleftarrow{\mathcal{D}}(\mathbf{1})_1 \stackrel{\text{def}}{=} \mathbf{1} \\
&\overleftarrow{\mathcal{D}}(\tau * \sigma)_1 \stackrel{\text{def}}{=} \overleftarrow{\mathcal{D}}(\tau)_1 * \overleftarrow{\mathcal{D}}(\sigma)_1 \\
&\overleftarrow{\mathcal{D}}(\tau \to \sigma)_1 \stackrel{\text{def}}{=} \Pi p : \overleftarrow{\mathcal{D}}(\tau)_1 . \Sigma p' : \overleftarrow{\mathcal{D}}(\sigma)_1 . \overleftarrow{\mathcal{D}}(\sigma)_2[{}^{p'}/_{p}] \multimap \overleftarrow{\mathcal{D}}(\tau)_2 \\
&\overleftarrow{\mathcal{D}}(\{\ell_1\tau_1 \mid \cdots \mid \ell_n\tau_n\})_1 \stackrel{\text{def}}{=} \left\{\ell_1\overleftarrow{\mathcal{D}}(\tau_1)_1 \mid \cdots \mid \ell_n\overleftarrow{\mathcal{D}}(\tau_n)_1\right\} \\
&\overleftarrow{\mathcal{D}}(\alpha)_1 \stackrel{\text{def}}{=} \alpha \\
&\overleftarrow{\mathcal{D}}(\mu\alpha.\tau)_1 \stackrel{\text{def}}{=} \mu\alpha.\overleftarrow{\mathcal{D}}(\tau)_1 \\
&\overleftarrow{\mathcal{D}}(\nu\alpha.\tau)_1 \stackrel{\text{def}}{=} \nu\alpha.\overleftarrow{\mathcal{D}}(\tau)_1
\end{aligned}$$

$$\begin{aligned}
&\overleftarrow{\mathcal{D}}(\mathbf{1})_2 \stackrel{\text{def}}{=} \underline{\mathbf{1}} \\
&\overleftarrow{\mathcal{D}}(\tau * \sigma)_2 \stackrel{\text{def}}{=} \overleftarrow{\mathcal{D}}(\tau)_2[{}^{\mathbf{fst}\,p}/_{p}] * \overleftarrow{\mathcal{D}}(\sigma)_2[{}^{\mathbf{snd}\,p}/_{p}] \\
&\overleftarrow{\mathcal{D}}(\tau \to \sigma)_2 \stackrel{\text{def}}{=} \Sigma p' : \overleftarrow{\mathcal{D}}(\tau)_1 . \overleftarrow{\mathcal{D}}(\sigma)_2[{}^{\mathbf{fst}\,(p\,p')}/_{p}] \\
&\overleftarrow{\mathcal{D}}(\{\ell_1\tau_1 \mid \cdots \mid \ell_n\tau_n\})_2 \stackrel{\text{def}}{=} \mathbf{case}\,p\,\mathbf{of}\,\{\ell_1 p \to \overleftarrow{\mathcal{D}}(\tau_1)_2 \mid \cdots \mid \ell_n p \to \overleftarrow{\mathcal{D}}(\tau_n)_2\} \\
&\overleftarrow{\mathcal{D}}(\alpha)_2 \stackrel{\text{def}}{=} \underline{\alpha}
\end{aligned}$$

$$\overleftarrow{\mathcal{D}}(\mu\alpha.\tau)_2 \stackrel{\mathrm{def}}{=} \underline{\nu\alpha}.\overleftarrow{\mathcal{D}}(\tau)_2[^{\mathbf{fold}\, p\, \mathbf{with}\, y \to \overleftarrow{\mathcal{D}}(\tau)_1[^{y \vdash \mathbf{roll}\, y}/_{\alpha}]}/_{p}]$$

$$\overleftarrow{\mathcal{D}}(\nu\alpha.\tau)_2 \stackrel{\mathrm{def}}{=} \underline{\mu\alpha}.\overleftarrow{\mathcal{D}}(\tau)_2[^{\mathbf{unroll}\, p}/_{p}]$$

For programs t, we define we define their efficient CHAD transformation $\overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(t)$ 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 primal and tangents in Appendix H):

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

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

$$\overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(\langle\rangle) \stackrel{\mathrm{def}}{=} \langle\langle\rangle, \underline{\lambda}\mathsf{v}.\underline{0}\rangle$$

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

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

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

$$\overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(\lambda x.t) \stackrel{\mathrm{def}}{=} \begin{array}{l}\mathbf{let}\, y = \lambda x.\overleftarrow{\mathcal{D}}_{\overline{\Gamma},x}(t)\, \mathbf{in} \\ \langle \lambda x.\mathbf{let}\, \langle z, z'\rangle = y\, x\, \mathbf{in}\, \langle z, \underline{\lambda}\mathsf{v}.\mathbf{snd}\, (z' \bullet \mathsf{v})\rangle, \\ \underline{\lambda}\mathsf{v}.\mathbf{case}\, \mathsf{v}\, \mathbf{of}\, !x \otimes \mathsf{v} \to \mathbf{fst}\, ((\mathbf{snd}\, (y\, x)) \bullet \mathsf{v})\rangle\end{array}$$

$$\overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(t\, s) \stackrel{\mathrm{def}}{=} \begin{array}{l}\mathbf{let}\, \langle x, x'\rangle = \overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(t)\, \mathbf{in} \\ \mathbf{let}\, \langle y, y'\rangle = \overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(s)\, \mathbf{in} \\ \langle \mathbf{fst}\, (x\, y), \underline{\lambda}\mathsf{v}.x' \bullet !y \otimes \mathsf{v} + y' \bullet (\mathbf{snd}\, (x\, y)) \bullet \mathsf{v}\rangle\end{array}$$

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

$$\overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(\mathbf{case}\, t\, \mathbf{of}\, \{\ell_1 x_1 \to s_1 \mid \cdots \mid \ell_n x_n \to s_n\}) \stackrel{\mathrm{def}}{=} \begin{array}{l}\mathbf{let}\, \langle y, y'\rangle = \overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(t)\, \mathbf{in} \\ \mathbf{case}\, y\, \mathbf{of}\, \{\ell_1 x_1 \to \\ \quad \mathbf{let}\, \langle z_1, z_1'\rangle = \overleftarrow{\mathcal{D}}_{\overline{\Gamma},x_1}(s_1)\, \mathbf{in} \\ \quad \langle z_1, \underline{\lambda}\mathsf{v}.\mathbf{let}\, \mathsf{v} = \mathbf{bunch}\, (y, \mathsf{v})\, \mathbf{of}\, \{ \\ \qquad\qquad (\ell_1 x_1, \mathsf{v}) \to z_1' \bullet \mathsf{v} \\ \qquad\qquad \mid (\ell_2 x_2, \mathsf{v}) \to \underline{0}\end{array}$$

$$
\begin{array}{l}
\quad\quad \mid \cdots \\
\quad\quad \mid (\ell_n x_n, \mathsf{v}) \to \underline{0}\} \,\mathbf{in} \\
\quad \mathbf{fst}\, \mathsf{v} + y' \bullet \mathbf{snd}\, \mathsf{v}\rangle \\
\quad \mid \cdots \mid \\
\quad \ell_n x_n \to \\
\mathbf{let}\, \langle z_n, z'_n\rangle = \overleftarrow{\mathcal{D}}_{\overline{\Gamma}, x_1}(s_1)\,\mathbf{in} \\
\langle z_n, \underline{\lambda}\mathsf{v}.\mathbf{let}\, \mathsf{v} = \mathbf{bunch}\,(y, \mathsf{v})\,\mathbf{of}\, \{ \\
\quad\quad (\ell_1 x_1, \mathsf{v}) \to \underline{0} \\
\quad\quad \mid \cdots \\
\quad\quad \mid (\ell_{n-1} x_{n-1}, \mathsf{v}) \to \underline{0} \\
\quad\quad \mid (\ell_n x_n, \mathsf{v}) \to z'_n \bullet \mathsf{v}\} \,\mathbf{in} \\
\quad \mathbf{fst}\, \mathsf{v} + y' \bullet \mathbf{snd}\, \mathsf{v}\rangle\}
\end{array}
$$

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

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

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

$$
\begin{array}{ll}
\overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(\mathbf{gen\, from}\, t\, \mathbf{with}\, x \to s) \stackrel{\text{def}}{=} & \mathbf{let}\, \langle y, y'\rangle = \overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(t)\,\mathbf{in} \\
& \mathbf{let}\, z = \lambda x.\overleftarrow{\mathcal{D}}_x(s)\,\mathbf{in} \\
& \langle \mathbf{gen\, from}\, y\, \mathbf{with}\, x \to \mathbf{fst}\,(z\, x) \\
& , \underline{\lambda}\mathsf{v}.y' \bullet \mathbf{fold}\, \mathsf{v}\, \mathbf{with}\, \mathsf{v} \to (\mathbf{snd}\,(z\, y)) \bullet \mathsf{v}\rangle
\end{array}
$$

6. Concrete denotational semantics

In order to proceed with our correctness proof of Automatic Differentiation, we need to establish the semantics of the program transformation in our setting. In this section, we construct denotational semantics for the target language.

6.1. Locally presentable categories and denotational model for the source language. We start by giving examples of concrete models for our source language. In order to do so, we show that any Cartesian closed locally presentable category yields a concrete model for the source language. Indeed, the only step needed to establish this fact is to recall that locally presentable categories have $\mu\nu$-polynomials [36, Theorem 3.7]. We recall below how to prove this result, taking the opportunity to recall some basic aspects on locally presentable categories.

The first fact to recall is that locally presentable categories are complete (besides being cocomplete by definition). Moreover:

Lemma 6.1. *Let $\mathcal{A}, \mathcal{B}$ be locally presentable categories.*

A functor $G : \mathcal{A} \to \mathcal{B}$ has a right adjoint if and only if G is accessible and preserves limits.

A functor $F : \mathcal{B} \to \mathcal{A}$ has a left adjoint if and only if F preserves colimits.

Lemma 6.2. *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 directed colimit, see [2].

If $\mathcal{A}$ is a locally presentable category, given an endofunctor $E : \mathcal{A} \to \mathcal{A}$, we have that E-CoAlg is locally presentable. Since the forgetful functor E-CoAlg $\to \mathcal{A}$ 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-CoAlg (terminal coalgebra of E), see [5]. ■

Proposition 6.3. *If $\mathcal{D}$ is locally presentable then $\mathcal{D}$ has $\mu\nu$-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 $\mu\nu\mathsf{Poly}_{\mathcal{D}}$ are locally presentable.

Given locally presentable categories $\mathcal{D}', \mathcal{D}''$, the projections $\pi_1 : \mathcal{D}' \times \mathcal{D}'' \to \mathcal{D}'$ and $\pi_2 : \mathcal{D}' \times \mathcal{D}'' \to \mathcal{D}''$ have right (and left) adjoints and, therefore, are accessible.

Moreover, given locally presentable categories $\mathcal{D}', \mathcal{D}'', \mathcal{D}'''$, if $E : \mathcal{D}' \to \mathcal{D}''$ and $J : \mathcal{D}' \to \mathcal{D}'''$ are accessible functors, then so is the induced functor $(E, J) : \mathcal{D}' \to \mathcal{D}'' \times \mathcal{D}'''$.

Furthermore, $\times : \mathcal{D} \times \mathcal{D} \to \mathcal{D}$ and $\sqcup : \mathcal{D} \times \mathcal{D} \to \mathcal{D}$ have, respectively, a left adjoint and a right adjoint. Therefore they are accessible.

Finally, by [36, Proposition 3.8], assuming their existence, μH and νH are accessible whenever $H : \mathcal{D}' \times \mathcal{D} \to \mathcal{D}$ is accessible and $\mathcal{D}'$ is locally presentable.

This completes the proof that all morphisms of $\mu\nu\mathsf{Poly}_{\mathcal{D}}$ are accessible. Hence, by Lemma 6.2, we have that all endofunctors in $\mu\nu\mathsf{Poly}_{\mathcal{D}}$ have initial algebras and terminal coalgebras. Therefore $\mathcal{D}$ has $\mu\nu$-polynomials. ∎

As a consequence, we have that *any locally presentable Cartesian closed category yields a concrete model for the source language.* In particular, **Set** yields a model for the source language and, thus, we obtain the following corollary by Corollary the universal property of the source language **Syn** (see Corollary 2.1):

Corollary 6.4 (Concrete semantics of the source language)**. Set** *is a bi-Cartesian closed category with $\mu\nu$-polynomials. Hence, once we fix the concrete semantics of the ground types and primitive operations of* **Syn** *by defining*

- *for each n-dimensional array* $\mathbf{real}^n \in \mathbf{Syn}$*,* $[\![\mathbf{real}^n]\!] \stackrel{\mathrm{def}}{=} \mathbb{R}^n \in \mathbf{Set}$ *in which $\mathbb{R}^n$ is the set underlying the n-dimensional Euclidean space;*
- *for each primitive* $\mathrm{op} \in \mathsf{Op}^m_{n_1,\dots,n_k}$*,* $[\![\mathrm{op}]\!] : \mathbb{R}^{n_1} \times \cdots \times \mathbb{R}^{n_k} \to \mathbb{R}^m$ *is the map in* **Set** *corresponding to the operation* op *intends to implement,*

we obtain a unique functor

$$[\![-]\!] : \mathbf{Syn} \to \mathbf{Set}$$

that extends these definitions to give a concrete denotational semantics for the entire source language such that $[\![-]\!]$ preserves the bi-Cartesian closed structure and the $\mu\nu$-polynomials.

6.2. Fam(Li) is bi-Cartesian closed and has $\mu\nu$-polynomials. Henceforth, we assume that T is an accessible monad on **Set**. We denote by **Li** the associated Eilenberg-Moore category. We further assume that **Li** has biproducts $(+, \mathbb{0})$. The main examples that we have in mind are the category of vector spaces **Li** = **Vect** and the category of commutative monoids **Li** = **CMon**.

We consider the indexed category

$$\begin{aligned} \mathbf{FLi}: \quad & \mathbf{Set}^{\mathrm{op}} && \to \mathbf{Cat} && (6.1)\\ & X && \mapsto \mathbf{Cat}\,[X, \mathbf{Li}] = \mathbf{Li}^X && (6.2)\\ & f : X \to Y && \mapsto \mathbf{Li}^f = \mathbf{Cat}\,[f, \mathbf{Li}] : \mathbf{Li}^Y \to \mathbf{Li}^X && (6.3)\end{aligned}$$

defined by the composition

$$\mathbf{Set}^{\mathrm{op}} \to \mathbf{Cat}^{\mathrm{op}} \xrightarrow{\mathbf{Cat}[-,\mathbf{Li}]} \mathbf{Cat} \tag{6.4}$$

in which $\mathbf{Cat}[-, \mathbf{Li}] = \mathbf{Li}^{(-)}$ is the exponential (internal hom) in $\mathbf{Cat}$. It is well known that

$$\Sigma_{\mathbf{Set}} \mathbf{FLi} \cong \mathbf{Fam}(\mathbf{Li}), \qquad \Sigma_{\mathbf{Set}} \mathbf{FLi}^{\mathrm{op}} \cong \mathbf{Fam}(\mathbf{Li}^{\mathrm{op}}) \tag{6.5}$$

where $\mathbf{Fam}(\mathbf{Li})$ and $\mathbf{Fam}(\mathbf{Li}^{\mathrm{op}})$ are, respectively, the free cocompletion under coproducts of $\mathbf{Li}$ and of $\mathbf{Li}^{\mathrm{op}}$. We have the following results.

Proposition 6.5. *For any category $\mathcal{D}$, $\mathbf{Fam}(\mathcal{D})$ and $\mathbf{Fam}(\mathcal{D}^{\mathrm{op}})$ are extensive.*

Proof: It follows from [10, Proposition 2.4]. ■

Proposition 6.6. *If $\mathcal{D}$ has biproducts $(+, \mathbb{0})$ and products, $\mathbf{Fam}(\mathcal{D})$ is Cartesian closed. Codually, if $\mathcal{C}$ has biproducts $(+, \mathbb{0})$ and coproducts, $\mathbf{Fam}(\mathcal{C}^{\mathrm{op}})$ is Cartesian closed.*

Proof: Namely, given families of objects $\mathcal{Y} : Y \to \mathcal{D}, \mathcal{Z} : Z \to \mathcal{D}$, we have that

$$\begin{array}{llll} (\mathcal{Y} \Rightarrow \mathcal{Z}) : & \mathbf{Fam}(\mathcal{D})(\mathcal{Y}, \mathcal{Z}) & \to & \mathcal{D} \\ & \left(g : Y \to Z, (\alpha_y : \mathcal{Y}(y) \to \mathcal{Z}(g(y)))_{y \in Y}\right) & \mapsto & \prod_{y \in Y} \mathcal{Z}(g(y)) \end{array}$$

is the exponential in $\mathbf{Fam}(\mathcal{D})$. Codually,

$$\begin{array}{llll} (\mathcal{Y} \Rightarrow \mathcal{Z}) : & \mathbf{Fam}(\mathcal{C}^{\mathrm{op}})(\mathcal{Y}, \mathcal{Z}) & \to & \mathcal{C} \\ & \left(g : Y \to Z, (\alpha_y : \mathcal{Z}(g(y)) \to \mathcal{Y}(y))_{y \in Y}\right) & \mapsto & \coprod_{y \in Y} \mathcal{Z}(g(y)) \end{array}$$

is the exponential in $\mathbf{Fam}(\mathcal{C}^{\mathrm{op}})$. ■

Proposition 6.7. *If $\mathcal{D}$ is locally presentable, $\mathbf{Fam}(\mathcal{D})$ and $\mathbf{Fam}(\mathcal{D}^{\mathrm{op}})$ are locally presentable.*

By Proposition 6.3, as a consequence of the above, we have:

Corollary 6.8. *The categories $\mathbf{Fam}(\mathbf{Li})$ and $\mathbf{Fam}(\mathbf{Li}^{\mathrm{op}})$ are locally presentable Cartesian closed categories. As a consequence, $\mathbf{Fam}(\mathbf{Li})$ and $\mathbf{Fam}(\mathbf{Li}^{\mathrm{op}})$ have $\mu\nu$-polynomials and yield models for the source language.*

Indeed, we prove below that $\mathbf{FLi} : \mathbf{Set}^{\mathrm{op}} \to \mathbf{Cat}$ yields a model for the target language. This also provides another proof that $\Sigma_{\mathbf{Set}}\mathbf{FLi} \cong \mathbf{Fam}(\mathbf{Li})$ and $\Sigma_{\mathbf{Set}}\mathbf{FLi}^{\mathrm{op}} \cong \mathbf{Fam}(\mathbf{Li}^{\mathrm{op}})$ are Cartesian closed categories with $\mu\nu$-polynomials by the results of Section 3.

6.3. Li is a Σ-bimodel for inductive and coinductive types. To show that $\mathbf{FLi} : \mathbf{Li}^{\mathrm{op}} \to \mathbf{Cat}$ yields a model for the target language, we first prove that $\mathbf{FLi} : \mathbf{Set}^{\mathrm{op}} \to \mathbf{Cat}$ is a Σ-bimodel for inductive and coinductive types. We already know that $\mathbf{Set}$ has $\mu\nu$-polynomials. Since $\mathbf{Li}$ is complete and cocomplete, we get that $\mathbf{FLi}\,(X) = \mathbf{Li}^X$ is complete and cocomplete (limits and colimits are constructed pointwise). In particular, $\mathbf{FLi}\,(X) = \mathbf{Li}^X$ has biproducts (also constructed pointwise) with zero objects. Moreover, for any function $f : X \to Y$ in $\mathbf{Set}$, we have that

$$\mathbf{Li}^f = \mathbf{FLi}\,(f) : \mathbf{Cat}\,[Y, \mathbf{Li}] \to \mathbf{Cat}\,[X, \mathbf{Li}] \tag{6.6}$$

has a (fully faithful) left adjoint and a (fully faithful) right adjoint, given by the left and right Kan extensions respectively. Namely, for each $\mathcal{X} : X \to \mathbf{Li}$,

$$\mathsf{ran}_f\mathcal{X}(x) = \prod_{i\in f^{-1}(x)} \mathcal{X}(i), \qquad\qquad \mathsf{lan}_f\mathcal{X}(x) = \coprod_{i\in f^{-1}(x)} \mathcal{X}(i). \tag{6.7}$$

Therefore, since $\mathbf{FLi}\,(f)$ is left and right adjoint, we get that it preserves limits, colimits (in particular, biproducts), initial algebras and terminal coalgebras (by Theorem 7.2). Indeed, $\mathbf{FLi}\,(f)$ strictly preserves biproducts (and zero object), initial algebras and terminal coalgebras, provided that we have chosen ones.

Since we have, of course, the isomorphisms

$$\begin{aligned}
\mathbf{FLi}\,(X \sqcup Y) &= \mathbf{Cat}\,[X \sqcup Y, \mathbf{Li}] \\
&\cong \mathbf{Cat}\,[X, \mathbf{Li}] \times \mathbf{Cat}\,[Y, \mathbf{Li}] \\
&= \mathbf{FLi}\,(X) \times \mathbf{FLi}\,(Y)
\end{aligned}$$

we have that $\mathbf{FLi}$ is extensive. Indeed, we have

$$\mathcal{S}^{(X,Y)} : \mathbf{FLi}\,(X) \times \mathbf{FLi}\,(Y) \to \mathbf{FLi}\,(X \sqcup Y) \tag{6.8}$$

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 6.1. *The strictly indexed category* **FLi** *is a* Σ*-bimodel for inductive and coinductive types. Therefore* $\Sigma_{\mathbf{Set}}\mathbf{FLi}$ *and* $\Sigma_{\mathbf{Set}}\mathbf{FLi}^{\mathrm{op}}$ *have* $\mu\nu$*-polynomials.*

Proof: It only remains to prove that all the endomorphisms in $\mu\nu\mathsf{Poly}_{\mathbf{FLi}}$ have initial algebras and terminal coalgebras. In order to do so, by Lemma 6.2, it is enough to prove that $\mu\nu\mathsf{Poly}_{\mathbf{FLi}}$ is a subcategory of the category of locally presentable categories and accessible functors between them.

As proved in Lemma 6.2, 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 **Li** is locally presentable. Also, since the biproduct $+ : \mathbf{Li}^X \times \mathbf{Li}^X \to \mathbf{Li}^X$ 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 $f : X \to Y$.

Finally, by [36, Proposition 3.8], assuming their existence, μh and νh are accessible whenever $h : \mathcal{D}' \times \mathcal{D} \to \mathcal{D}$ 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 $\mu\nu\mathsf{Poly}_{\mathbf{FLi}}$ are accessible functors between locally presentable categories and, hence, any endomorphism $\mu\nu\mathsf{Poly}_{\mathbf{FLi}}$ has initial algebra and terminal coalgebra. ∎

6.4. Li yields a model for the target language. It remains only to prove that **Li** is a Σ-bimodel for function types.

The model of Cartesian dependent type theory associated is, of course, the strictly indexed category $\mathbf{FSet} : \mathbf{Set}^{\mathrm{op}} \to \mathbf{Cat}$

$$X \mapsto \mathbf{Cat}\left[X, \mathbf{Li}\right].$$

The fact that **FSet** satisfies full, faithful, democratic comprehension with Π-types and strong Σ-types is well-known [19].

The fact that **Li** has Π- types follows, for instance, from [38, Theorem 5.2.9] while the Σ-types and $\multimap$ follows from [38, Theorem 5.6.3].

6.5. The denotational model for the target language. We specialize further to a concrete denotational model for the target language in terms of real vector spaces. We can phrase of our specifications for the correctness proofs of Section 8 with respect to this concrete semantics. Namely, we consider $\mathbf{Li} = \mathbf{Vect}$, denoting by

$$\mathbf{FVect} : \mathbf{Set}^{\mathrm{op}} \to \mathbf{Cat}, \qquad \mathbf{FVect}(X) = \mathbf{Vect}^X$$

the associated indexed category. Recall that we have that

$$\Sigma_{\mathbf{Set}}\mathbf{FVect} \cong \mathbf{Fam}(\mathbf{Vect}), \qquad \Sigma_{\mathbf{Set}}\mathbf{FVect}^{\mathrm{op}} \cong \mathbf{Fam}(\mathbf{Vect}^{\mathrm{op}}) \tag{6.9}$$

where $\mathbf{Fam}(\mathbf{Vect})$ and $\mathbf{Fam}(\mathbf{Vect}^{\mathrm{op}})$ are, respectively, the free cocompletion under coproducts of of $\mathbf{Vect}$ and $\mathbf{Vect}^{\mathrm{op}}$.

As proved above, seeing that $\mathbf{Vect}$ is locally presentable, $\mathbf{FVect}$ is a Σ-bimodel for tuple, function, sum, inductive and coinductive types and, hence, it provides a suitable model for our target language. Once we specialize from arbitrary families $\mathbf{FLi}$ in a locally indexed category to $\mathbf{FVect}$, there is a bit more we can say about the interpretation of type formers.

Product structure. Assume that $(M, R), (N, V)$ are objects of $\Sigma_{\mathbf{Set}}\mathbf{FVect}$ (or $\Sigma_{\mathbf{Set}}\mathbf{FVect}^{\mathrm{op}}$). By Propositions 3.2 and 3.1, we have that

$$(M, R) \times (N, V) = (M \times N, (i, j) \mapsto R(i) \times V(j))\,. \tag{6.10}$$

gives the product of (M, R) and (N, V) in $\Sigma_{\mathbf{Set}}\mathbf{FVect}$ (and in $\Sigma_{\mathbf{Set}}\mathbf{FVect}^{\mathrm{op}}$). The terminal object in $\Sigma_{\mathbf{Set}}\mathbf{FVect}$ (and in $\Sigma_{\mathbf{Set}}\mathbf{FVect}^{\mathrm{op}}$) is given by $(\mathbb{1}, \mathbb{0})$.

Coproduct structure. Assume that $(M, R), (N, V)$ are objects of $\Sigma_{\mathbf{Set}}\mathbf{FVect}$ (or $\Sigma_{\mathbf{Set}}\mathbf{FVect}^{\mathrm{op}}$). By Corollaries 3.7 and 3.6, we have that

$$(M, R) \sqcup (N, V) = (M \sqcup N, \langle R, V\rangle : M \sqcup N \to \mathbf{Set})\,. \tag{6.11}$$

gives the coproduct of (M, R) and (N, V) in $\Sigma_{\mathbf{Set}}\mathbf{FVect}$ (and in $\Sigma_{\mathbf{Set}}\mathbf{FVect}^{\mathrm{op}}$). The initial objects are given by $(\emptyset, \mathbb{0})$.

Lists and Streams. The categories $\Sigma_{\mathbf{Set}}\mathbf{FVect}$ and $\Sigma_{\mathbf{Set}}\mathbf{FVect}^{\mathrm{op}}$ have $\mu\nu$-polynomials by Corollary 3.20 since $\mathbf{FVect}$ is Σ-bimodel for inductive and coinductive types by Theorem 6.1. But, in fact, as showed above $\mathbf{Fam}(\mathbf{Vect}) \cong \mathbf{Fam}(\mathbf{Vect})$ and $\mathbf{Fam}(\mathbf{Vect}^{\mathrm{op}}) \cong \Sigma_{\mathbf{Set}}\mathbf{FVect}^{\mathrm{op}}$ have $\mu\nu$-polynomials because they are locally presentable categories (Corollary 6.8).

Therefore, since we are in the locally presentable setting, we can compute the initial algebras or terminal coalgebras of $\mu\nu$-polynomials via directed colimits and limits (see [2]). In particular, denoting by $\underline{V} : V \to \mathbf{Vect}$ the functor constantly equal to the vector space V (in which, by abuse of language V is also the set underlying the vector space), given the endofunctors

$$E : \mathbf{Fam}(\mathbf{Vect}) \to \mathbf{Fam}(\mathbf{Vect}), \qquad\qquad H : \mathbf{Fam}(\mathbf{Vect}) \to \mathbf{Fam}(\mathbf{Vect})$$

in which $E(X,x) = (\mathbb{1}, \mathbb{0}) \sqcup (X,x) \times (V, \underline{V})$ and $H(X,x) = (X,x) \times (V, \underline{V})$, we have that

$$\mu E = \coprod_{i=0}^{\infty} (V, \underline{V})^n, \qquad \nu H = \prod_{i=0}^{\infty} (V, \underline{V}),$$

in $\mathbf{Fam}(\mathbf{Vect})$ and $\mathbf{Fam}(\mathbf{Vect}^{\mathrm{op}})$.

For the case of μE, this means that

$$\mathsf{List}\,(V, \underline{V}) = \mu E : \coprod_{i=0}^{\infty} V^n \to \mathbf{Vect}$$

is induced/defined by the functors $\underline{V^n} : V^n \to \mathbf{Vect}$ constantly equal to the product V^n (in $\mathbf{Vect}$) in each component V^n of the set $\coprod_{i=0}^{\infty} V^n$ (in both $\mathbf{Fam}(\mathbf{Vect})$ and $\mathbf{Fam}(\mathbf{Vect}^{\mathrm{op}})$).

For the case of νE, this means that

$$\mathsf{Stream}\,(V, \underline{V}) : \prod_{i=0}^{\infty} V \to \mathbf{Vect}$$

is the functor constantly equal to the product $\displaystyle\prod_{i=0}^{\infty} V$ in the case of $\mathbf{Fam}(\mathbf{Vect})$, while it is the functor constantly equal to $\displaystyle\coprod_{i=0}^{\infty} V$ in the case of $\mathbf{Fam}(\mathbf{Vect}^{\mathrm{op}})$.

Primitive types and operations. The reason we work with a concrete semantics in terms of (families of) real vector spaces is because they suffice to interpret spaces of (co)tangent vectors as well as (transposed) derivatives of smooth functions. In particular, we have a constant family of vector spaces $\underline{\mathbb{R}}^n$ to interpret $\underline{\mathbf{real}}^n$ and, for every smooth function $[\![\mathrm{op}]\!]$ that op is tended to implement, we have linear functions $D[\![\mathrm{op}]\!]$ and $D[\![\mathrm{op}]\!]^t$ (the usual mathematical derivative and transposed derivative of $[\![\mathrm{op}]\!]$!) to interpret $D\mathrm{op}$ and $D\mathrm{op}^t$. Fixing these choices if enough to give a full denotational semantics to the target language of CHAD.

Corollary 6.9 (Concrete semantics of the target language)**.** **FVect** *is a* Σ*-bimodel for tuples, function types, sum types, inductive and coinductive types. Hence, considering the* Σ*-bimodel of primitive types and operations defined by*

(1) for each n*-dimensional array* $\mathbf{real}^n \in \mathbf{Syn}$*,* $C_{\mathbf{real}^n} = [\![\mathbf{real}^n]\!] \stackrel{\text{def}}{=} \mathbb{R}^n \in \mathbf{Set}$ *in which* $\mathbb{R}^n$ *is the set underlying the* n*-dimensional Euclidean space;*

(2) for each n*-dimensional array* $\mathbf{real}^n \in \mathbf{Syn}$*,*

$$L_{\mathbf{real}^n}, L'_{\mathbf{real}^n} \in \mathbf{FVect}\,(\mathbb{R}^n)$$

in which $L_{\mathbf{real}^n} = L'_{\mathbf{real}^n} = \underline{\mathbb{R}}^n : \mathbb{R}^n \to \mathbf{Vect}$ *is the functor/family constantly equal to the* n*-dimensional Euclidean space;*

(3) for each primitive $\mathrm{op} \in \mathsf{Op}^m_{n_1,\ldots,n_k}$*:*

(a) $f_{\mathrm{op}} = [\![\mathrm{op}]\!] : \mathbb{R}^{n_1} \times \cdots \times \mathbb{R}^{n_k} \to \mathbb{R}^m$ *is the map in* **Set** *corresponding to the operation that* op *intends to implement (hence smooth when considered as a function between the Euclidean spaces* $\mathbb{R}^{n_1} \times \cdots \times \mathbb{R}^{n_k}$ *and* $\mathbb{R}^m$*);*

(b) $g_{\mathrm{op}} = [\![D\mathrm{op}]\!] = D[\![\mathrm{op}]\!] \in \mathbf{FVect}(\mathbb{R}^{n_1} \times \cdots \times \mathbb{R}^{n_k})(\underline{\mathbb{R}}^{n_1} \times \cdots \times \underline{\mathbb{R}}^{n_k}, \underline{\mathbb{R}}^m)$ *is the family of linear transformations corresponding to the derivative of the operation that* op *intends to implement;*

(c) $g'_{\mathrm{op}} = [\![D\mathrm{op}^t]\!] = D[\![\mathrm{op}]\!]^t \in \mathbf{FVect}(\mathbb{R}^{n_1} \times \cdots \times \mathbb{R}^{n_k})(\underline{\mathbb{R}}^m, \underline{\mathbb{R}}^{n_1} \times \cdots \times \underline{\mathbb{R}}^{n_k})$ *is the family of linear transformations corresponding to the transposed derivative of the operation that* op *intends to implement;*

we obtain, by Corollary 4.2, canonical functors

$$\Sigma_{[\![-]\!]}[\![-]\!] : \Sigma_{\mathbf{CSyn}}\mathbf{LSyn} \to \Sigma_{\mathbf{Set}}\mathbf{FVect} \cong \mathbf{Fam}(\mathbf{Vect})$$
$$\Sigma_{[\![-]\!]}[\![-]\!]^{op} : \Sigma_{\mathbf{CSyn}}\mathbf{LSyn}^{op} \to \Sigma_{\mathbf{Set}}\mathbf{FVect}^{op} \cong \mathbf{Fam}(\mathbf{Vect}^{\mathrm{op}}).$$

that extend these definitions to give a concrete denotational semantics for the entire target language of the forward AD and the reverse AD respectively such that $\Sigma_{[\![-]\!]}[\![-]\!]$ *and* $\Sigma_{[\![-]\!]}[\![-]\!]^{op}$ *are bi-Cartesian closed functors that preserve* $\mu\nu$*-polynomials.*

7. Sconing

Given a functor $G : \mathcal{C} \to \mathcal{D}$, the comma category $\mathcal{D} \downarrow G$ is also known as the *scone* or *Artin glueing* of G. Explicitly, the scone's objects are triples $(C_0 \in \mathcal{D}, C_1 \in \mathcal{C}, f : C_0 \to G(C_1))$ in which f is a morphism of $\mathcal{D}$. Its morphisms $(C_0, C_1, f) \to (C_0', C_1', f')$ are pairs $(h_0 : C_0 \to C_0', h_1 : C_1 \to C_1')$ such that

$$\begin{tikzcd} C_0 \arrow[r, "h_0"] \arrow[d, "f"'] & C_0' \arrow[d, "f'"] \\ G(C_1) \arrow[r, "G(h_1)"'] & G(C_1') \end{tikzcd}$$

commutes in $\mathcal{D}$.

The scone $\mathcal{D} \downarrow G$ inherits much of the structure of $\mathcal{D} \times \mathcal{C}$. For that reason, under suitable conditions, sconing can be seen as a 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 8 and, also, in [40, Section 8].

Under suitable conditions, the nice properties of $\mathcal{D} \downarrow G$ can be seen as consequences of the fact that the forgetful functor

$$\mathsf{L} : \mathcal{D} \downarrow G \to \mathcal{D} \times \mathcal{C}, \tag{7.1}$$

defined by $(C_0 \in \mathcal{D}, C_1 \in \mathcal{C}, f : C_0 \to G(C_1)) \mapsto (C_0, C_1)$, is comonadic and, in our case, even monadic. More precisely:

Theorem 7.1. *If $\mathcal{D}$ has binary products, then* (7.1) *is comonadic.*

Proof: By the universal property of comma categories, a diagram $D : \mathcal{S} \to \mathcal{D} \downarrow G$ corresponds biunivocally with triples

$$(D_0 : \mathcal{S} \to \mathcal{D}, D_1 : \mathcal{S} \to \mathcal{C}, \mathfrak{d} : D_0 \to GD_1) \tag{7.2}$$

in which D_0, D_1 are diagrams and $\mathfrak{d}$ 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

$$\left(\lim D_0, \lim D_1, \lim D_0 \xrightarrow{\mathsf{d}} \lim\, (G \circ D_1) \xrightarrow{\cong} G\left(\lim D_1\right)\right), \tag{7.3}$$

is the limit of D in $\mathcal{D} \downarrow G$, in which d is the morphism induced by the natural transformation $\mathfrak{d}$.

Now, given a diagram $D : \mathcal{S} \to \mathcal{D} \downarrow G$ such that $\mathsf{L} \circ D = (D_0, D_1) : \mathcal{S} \to \mathcal{D} \times \mathcal{C}$ 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 (7.3). Thus it is preserved by L. Since (7.1) is conservative, this completes the proof that (7.1) creates absolute limits.

Finally, since (7.1) has a right adjoint defined by

$$(Y, X) \mapsto (Y \times G(X), X, \pi_2 : Y \times G(X) \to G(X)),$$

the proof that (7.1) is comonadic is complete by Beck's Monadicity Theorem.*

■

Remark 7.1. *If $\mathcal{C}$ has a terminal object and $\mathcal{D}$ has binary products as above,* (7.1) *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 7.2. *Assume that $\mathcal{C}$ has binary coproducts and $\mathcal{D}$ has binary products. We have that* (7.1) *is comonadic and monadic provided that G has a left adjoint F.*

Proof: Firstly, of course, by Theorem 7.1, we have that (7.1) is comonadic. Secondly, by the dual of Theorem 7.1, we have that the forgetful functor $F \downarrow \mathcal{C} \to \mathcal{C} \times \mathcal{D}$ is monadic. Hence, since

$$\begin{tikzcd}
\mathcal{D} \downarrow G \arrow[rr, "\mathsf{L}"] \arrow[dr, leftrightarrow, "\cong"'] & & \mathcal{D} \times \mathcal{C} \\
& F \downarrow \mathcal{C} \arrow[ur] &
\end{tikzcd}$$

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 (7.1), that is to say, Corollary 7.2.†

7.1. Bi-Cartesian structure. The bi-Cartesian closed structure of the scone $\mathcal{D} \downarrow G$ follows from the well known result about monadic functors and creation of limits. Namely:

Proposition 7.3. *Monadic functors create all limits. Dually, comonadic functors create all colimits.*

*Despite its usual statement in terms of split (co)equalizers, Beck's Monadicity Theorem implies that: a left adjoint functor is comonadic if and only if it creates absolute limits. See, for instance, [29, pag. 550].

†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 to our proof and many others cases of interest.

Proof: See, for instance, [30, Section 1.4]. ■

As a corollary, then, we have the following explicit constructions.

Corollary 7.4. *Assuming that $\mathcal{C}$ and $\mathcal{D}$ have finite products and coproducts, if $G : \mathcal{C} \to \mathcal{D}$ has a left adjoint, then $\mathsf{L} : \mathcal{D} \downarrow G \to \mathcal{D} \times \mathcal{C}$ creates limits and colimits. In particular, $\mathcal{D} \downarrow G$ is bi-Cartesian and, in case $\mathcal{D} \times \mathcal{C}$ is a distributive category, so is $\mathcal{D} \downarrow G$.*

Proof: Given a diagram $D : \mathcal{S} \to \mathcal{D} \downarrow G$, we have that it is uniquely determined by a triple $(D_0 : \mathcal{S} \to \mathcal{D}, D_1 : \mathcal{S} \to \mathcal{C}, \mathfrak{d} : D_0 \to GD_1)$ like in (7.2). In this case, we have that:

(1) In the proof of Theorem 7.1, 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 so all the limits are created like (7.4).

More precisely, in this case, assuming that $\mathsf{L} \circ D = (D_0, D_1) : \mathcal{S} \to \mathcal{D} \times \mathcal{C}$ has a limit, we get that both $\lim D_0$ and $\lim D_1$ exist, since the projections $\mathcal{D} \times \mathcal{C} \to \mathcal{D}$ and $\mathcal{D} \times \mathcal{C} \to \mathcal{C}$ 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

$$\left(\lim D_0, \lim D_1, \lim D_0, \xrightarrow{\mathsf{d}} \lim\ (G \circ D_1) \xrightarrow{\cong} G\,(\lim D_1)\right), \tag{7.4}$$

like in (7.3), in which d is the morphism induced by $\mathfrak{d}$ and $\lim\ (G \circ D_1) \cong G\,(\lim D_1)$ comes from the fact that G preserves limits.

(2) Assuming that $\mathsf{L} \circ D = (D_0, D_1) : \mathcal{S} \to \mathcal{D} \times \mathcal{C}$ has a limit, we get that both $\operatorname{colim} D_0$ and $\operatorname{colim} D_1$ exist. In this case, the colimit of D is given by

$$\left(\operatorname{colim} D_0, \operatorname{colim} D_1, \operatorname{colim} D_0 \xrightarrow{\mathsf{d}} \operatorname{colim}\ (G \circ D_1) \to G\,(\operatorname{colim} D_1)\right), \tag{7.5}$$

in which $\operatorname{colim}\ (G \circ D_1) \to G\,(\operatorname{colim} D_1)$ is the induced comparison.

■

7.2. Closed structure. 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 7.5. *If a category is monadic-comonadic over a finitely complete cartesian closed category, then it is finitely complete cartesian closed as well.*

Proof: See, for instance, a slightly more general version in [27, Theorem 1.8.2]. Indeed, assuming that $G : \mathcal{C} \to \mathcal{D}$ 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).

Given an object $W \in \mathcal{C}$, we have an isomorphism

$$\begin{tikzcd}
\mathcal{C} \arrow[r, "(W\times -)"] \arrow[d, "G"'] \arrow[dr, phantom, "\cong"] & \mathcal{C} \arrow[d, "G"] \\
\mathcal{D} \arrow[r, "(G(W)\times -)"'] & \mathcal{D}
\end{tikzcd} \tag{7.6}$$

in which we know that $(W \times F(-)) \dashv (G(W) \times G(-))$. Since $\mathcal{C}$ has equalizers and G is comonadic, we get that $(W \times -)$ has a right adjoint by Dubuc's adjoint triangle theorem.* ∎

Explicitly, we get:

Corollary 7.6. *Let $\mathcal{C}$ and $\mathcal{D}$ be finitely complete cartesian closed categories. If $G : \mathcal{C} \to \mathcal{D}$ has a left adjoint, we get that $\mathcal{D} \downarrow G$ is finitely complete cartesian closed. More precisely:*

$$(C_0, C_1, f) \Rightarrow (D_0, D_1, f') = (P, C_1 \Rightarrow D_1, f \Rightarrow f'),$$

where we write $f \Rightarrow f'$ for the following pullback:

$$\begin{tikzcd}
P \arrow[rr] \arrow[d, "f \Rightarrow f'"'] & & C_0 \Rightarrow D_0 \arrow[d, "C_0 \Rightarrow f'"] \\
G(C_1 \Rightarrow D_1) \arrow[r] & G(C_1) \Rightarrow G(D_1) \arrow[r, "f \Rightarrow G(D_1)"'] & C_0 \Rightarrow G(D_1)
\end{tikzcd} \tag{7.7}$$

*See [12] or, for instance, [26, Corollary 1.2] for the precise statement in our case.

7.3. Initial algebras and final coalgebras. Recall the definition of preservation, reflection and creation of initial algebras and final coalgebras (see Definitions E.3 and E.7). We prove and establish the result that says that monadic functors create initial algebras, while, dually, comonadic functors create final coalgebras.

We firstly establish the fact that left adjoint functors preserve initial algebras and, dually, right adjoint functors preserve final coalgebras. In order to do so, we start by observing that:

Lemma 7.7. *Let*

$$\begin{tikzcd} \mathcal{C} \arrow[rr, bend right, "G"'] \arrow[rr, phantom, "\perp(\varepsilon,\eta)"] & & \mathcal{D} \arrow[ll, bend right, "F"'] \end{tikzcd}$$

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*

$$\begin{tikzcd} E\text{-Alg} \arrow[rr, bend right, "\underline{\hat{G}}_\gamma"'] \arrow[rr, phantom, "\perp(\underline{\hat{\varepsilon}},\underline{\hat{\eta}})"] & & E'\text{-Alg} \arrow[ll, bend right, "\check{F}_\gamma"'] \end{tikzcd} \tag{7.8}$$

in which $\check{F}_\gamma$ *is defined as in E.3 and* $\underline{\hat{G}}_\gamma$ *is defined as follows:*[*]

$$\begin{aligned} \underline{\hat{G}}_\gamma : \ E'\text{-Alg} &\to E\text{-Alg} \\ (Y,\xi) &\mapsto \left(G(Y), G(\xi) \circ GE(\varepsilon_Y) \circ G\left(\gamma^{-1}_{G(Y)}\right) \circ \eta_{E'G(Y)}\right) \\ f &\mapsto G(f). \end{aligned}$$

[*]The right adjoint $\underline{\hat{G}}_\gamma$ does not come out of the blue. For the reader who knows a bit of 2-dimensional category theory, it is interesting to note that the association $(F,\gamma) \mapsto \check{F}_\gamma$ of Lemma E.1 is actually part of a 2-functor. The codomain of this 2-functor is **Cat**, while the domain is the 2-category of endomorphisms in **Cat**, lax natural transformations and modifications.

By doctrinal adjunction (see [21] or, for instance, [27, Corollary 1.4.15]), whenever (F,γ) is pseudonatural (meaning that γ is invertible) and F has a right adjoint in **Cat**, the pair (F,γ) has a right adjoint

$$\left(G, (GE\varepsilon) \cdot \left(G\gamma_G^{-1}\right) \cdot (\eta_{E'G})\right)$$

in the 2-category of endofunctors. Therefore, since 2-functors preserve adjunctions, we get that $\check{F}_\gamma$ has a right adjoint given by $\check{G}_{(GE\varepsilon)\cdot\left(G\gamma_G^{-1}\right)\cdot(\eta_{E'G})}$, denoted herein by $\underline{\hat{G}}_\gamma$, whenever γ is invertible and F has a right adjoint.

Proof: In fact, the counit and unit, $\underline{\hat{\varepsilon}}, \underline{\hat{\eta}}$, are defined pointwise by the original counit and unit. That is to say, $\underline{\hat{\varepsilon}}_{(Y,\xi)} = \varepsilon_Y$ and $\underline{\hat{\eta}}_{(W,\zeta)} = \eta_W$. ∎

Dually, we get:

Lemma 7.8. *Let*

$$\begin{tikzcd} \mathcal{C} \arrow[rr, bend right, "G"'] \arrow[rr, phantom, "\perp(\varepsilon,\eta)"] & & \mathcal{D} \arrow[ll, bend right, "F"'] \end{tikzcd}$$

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

$$\begin{tikzcd} E\text{-CoAlg} \arrow[rr, bend right, "\tilde{G}^{\beta}"'] \arrow[rr, phantom, "\perp(\hat{\varepsilon},\hat{\eta})"] & & E'\text{-CoAlg} \arrow[ll, bend right, "\hat{F}^{\beta}"'] \end{tikzcd} \qquad (7.9)$$

in which $\tilde{G}^{\beta}$ is defined as in E.7 and $\hat{F}^{\beta}$ is defined as follows:

$$\begin{aligned} \hat{F} : \ E'\text{-CoAlg} &\to E\text{-CoAlg} \\ (W,\zeta) &\mapsto \Big(W, \varepsilon_{EF(W)} \circ F(\beta^{-1}_{F(W)}) \circ FE'\left(\eta_W\right) \circ F\left(\zeta\right)\Big) \\ g &\mapsto F(g). \end{aligned}$$

As an immediate consequence, we have that:

Theorem 7.2. *Right adjoint functors preserve terminal coalgebras. Dually, left adjoints preserve initial algebras.*

Proof: Let $G : \mathcal{C} \to \mathcal{D}$ 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 $\tilde{G}^{\beta} : E\text{-CoAlg} \to E'\text{-CoAlg}$ (as defined in E.7) has a left adjoint by Lemma 7.8. Therefore $\tilde{G}^{\beta}$ preserves limits and, in particular, terminal objects. This completes the proof that G preserves final coalgebras (see Definition E.7). ∎

Finally, we can state the result about monadic functors. Namely:

Theorem 7.3. *Monadic functors create final coalgebras. Dually, comonadic functors create initial algebras.*

Proof: Let $G : \mathcal{C} \to \mathcal{D}$ 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 $\tilde{G}^{\beta} : E\text{-CoAlg} \to E'\text{-CoAlg}$ (as defined in E.7) has a left adjoint by Lemma 7.8. Moreover, we have the commutative diagram

$$\begin{tikzcd} E\text{-CoAlg} \arrow[r, "\tilde{G}^{\beta}"] \arrow[d] & E'\text{-CoAlg} \arrow[d] \\ \mathcal{C} \arrow[r, "G"'] & \mathcal{D} \end{tikzcd} \tag{7.10}$$

in which the vertical arrows are the forgetful functors.

Since we know that all the functors in (7.10) but $\tilde{G}^{\beta}$ create absolute colimits, we conclude that $\tilde{G}^{\beta}$ creates absolute colimits as well. Therefore $\tilde{G}^{\beta}$ is monadic and, thus, it creates all limits. In particular, $\tilde{G}^{\beta}$ creates terminal objects. This completes the proof that G creates final coalgebras (see Definition E.7). ∎

7.4. $\mu\nu$-polynomials. Finally, we can establish the existence of $\mu\nu$-polynomials in the scone, and the preservation of the initial algebras and final coalgebras by the forgetful functor.

Corollary 7.9. *Monadic-comonadic functors create $\mu\nu$-polynomials. More precisely, if $G : \mathcal{A} \to \mathcal{B}$ is monadic-comonadic and $\mathcal{B}$ has $\mu\nu$-polynomials, then*

(1) $\mathcal{A}$ has $\mu\nu$-polynomials;
(2) for each $\mu\nu$-polynomial endofunctor E on $\mathcal{A}$, there is a $\mu\nu$-polynomial endofunctor $\underline{E}$ on $\mathcal{B}$ such that $G \circ E \cong \underline{E} \circ G$ (and G creates the initial algebra and the terminal coalgebra of E).

Proof: Let $G : \mathcal{A} \to \mathcal{B}$ be a monadic-comonadic functor in which $\mathcal{B}$ has $\mu\nu$-polynomials. We inductively define the set $\underline{\overline{\times G}}$ as follows:

$\underline{\overline{\times G}}$1. the identity functor $\mathbb{1} \to \mathbb{1}$ belongs to $\underline{\overline{\times G}}$;
$\underline{\overline{\times G}}$2. $G : \mathcal{A} \to \mathcal{B}$ belongs to $\underline{\overline{\times G}}$;
$\underline{\overline{\times G}}$3. if $G' : \mathcal{A}' \to \mathcal{B}'$ and $G'' : \mathcal{A}'' \to \mathcal{B}''$ belong $\underline{\overline{\times G}}$, then so does the product $G' \times G'' : \mathcal{A}' \times \mathcal{A}'' \to \mathcal{B}' \times \mathcal{B}''$.

We have a clear bijection dom between $\underline{\overline{\times G}}$ and the objects of $\mu\nu\mathsf{Poly}_{\mathcal{A}}$ defined inductively by

$$\mathrm{dom}\,(\mathbb{1} \to \mathbb{1}) = \mathbb{1}\ , \qquad \mathrm{dom}\,(G) = \mathcal{A},$$
$$\mathrm{dom}\,(G' \times G'') = \mathrm{dom}\,(G') \times \mathrm{dom}\,(G'').$$

In other words, the function $\mathrm{dom} : \overline{\underline{\times G}} \to \mathrm{obj}\,(\mu\nu\mathsf{Poly}_{\mathcal{A}})$ gives the domain of each functor in $\overline{\underline{\times G}}$. Analogously, we define the bijection $\mathrm{codom} : \overline{\underline{\times G}} \to \mathrm{obj}\,(\mu\nu\mathsf{Poly}_{\mathcal{B}})$ which gives the codomain of each functor in $\overline{\underline{\times G}}$.

Since we know that G creates initial algebras and final coalgebras, to complete the proof, it is enough to show that, for any $\mu\nu$-polynomial $H : \mathcal{C} \to \mathcal{D}$ in $\mu\nu\mathsf{Poly}_{\mathcal{A}}$, there is a morphism $\overline{\underline{H}}$ of $\mu\nu\mathsf{Poly}_{\mathcal{B}}$ such that there is an isomorphism

$$\begin{tikzcd}
\mathcal{C} \arrow[rr, "H"] \arrow[d, "\mathrm{dom}^{-1}(\mathcal{C})"'] & & \mathcal{D} \arrow[d, "\mathrm{dom}^{-1}(\mathcal{D})"] \\
\overline{\underline{\mathcal{C}}} \arrow[rr, "\overline{\underline{H}}"'] \arrow[urr, phantom, "\overset{\cong}{\leftrightarrow}"] & & \overline{\underline{\mathcal{D}}}
\end{tikzcd} \tag{7.11}$$

where $\overline{\underline{\mathcal{D}}} := \mathrm{codom} \circ \mathrm{dom}^{-1}(\mathcal{D})$ and $\overline{\underline{\mathcal{D}}} := \mathrm{codom} \circ \mathrm{dom}^{-1}(\mathcal{C})$.

We start by proving that the objects of $\mu\nu\mathsf{Poly}_{\mathcal{A}}$ 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

$$\mathrm{id}_{\overline{\underline{\mathcal{C}}}} \circ \mathrm{dom}^{-1}(\mathcal{C}) = \mathrm{dom}^{-1}(\mathcal{C}) \circ \mathrm{id}_{\mathcal{C}}$$

for any given object $\mathcal{C}$ of $\mu\nu\mathsf{Poly}_{\mathcal{A}}$. Moreover, given morphisms $J : \mathcal{D}'' \to \mathcal{D}'''$ and $E : \mathcal{D}' \to \mathcal{D}''$ of $\mu\nu\mathsf{Poly}_{\mathcal{A}}$ such that we have natural isomorphisms

$$\begin{aligned}
\gamma :\ & \overline{\underline{E}} \circ \mathrm{dom}^{-1}(\mathcal{D}') \ \cong \mathrm{dom}^{-1}(\mathcal{D}'') \circ E \\
\gamma' :\ & \overline{\underline{J}} \circ \mathrm{dom}^{-1}(\mathcal{D}'') \ \cong \mathrm{dom}^{-1}(\mathcal{D}''') \circ J
\end{aligned}$$

in which $\overline{\underline{J}}$ and $\overline{\underline{E}}$ are morphisms of $\mu\nu\mathsf{Poly}_{\mathcal{B}}$, we have that

$$\begin{tikzcd}
\mathcal{D}' \arrow[rr, "E"] \arrow[d, "\mathrm{dom}^{-1}(\mathcal{D}')"'] & & \mathcal{D}'' \arrow[rr, "J"] \arrow[d, "\mathrm{dom}^{-1}(\mathcal{D}'')"] & & \mathcal{D}''' \arrow[d, "\mathrm{dom}^{-1}(\mathcal{D}''')"] \\
\overline{\underline{\mathcal{D}'}} \arrow[rr, "\overline{\underline{E}}"'] \arrow[urr, phantom, "\overset{\gamma}{\leftrightarrow}"] & & \overline{\underline{\mathcal{D}''}} \arrow[rr, "\overline{\underline{J}}"'] \arrow[urr, phantom, "\overset{\gamma'}{\leftrightarrow}"] & & \overline{\underline{\mathcal{D}'''}}
\end{tikzcd} \tag{7.12}$$

is a natural isomorphism and $\overline{\underline{J}} \circ \overline{\underline{E}}$ is a morphism in $\mu\nu\mathsf{Poly}_{\mathcal{B}}$.

Finally, we complete the proof that all the morphisms of $\mu\nu\mathsf{Poly}_{\mathcal{A}}$ satisfy the property above by proving by induction over the Definition 1.1 of $\mu\nu\mathsf{Poly}_{\mathcal{A}}$.

M1. for any object $\mathcal{C}$ of $\mu\nu\mathsf{Poly}_{\mathcal{A}}$, the unique functor $\mathcal{C} \to \mathbb{1}$ is such that

$$\begin{tikzcd} \mathcal{C} \arrow[r] \arrow[d, "\mathrm{dom}^{-1}(\mathcal{C})"'] & \mathbb{1} \arrow[d, "\mathrm{dom}^{-1}(\mathbb{1})"] \\ \underline{\overline{\mathcal{C}}} \arrow[r] & \mathbb{1} \end{tikzcd} \tag{7.13}$$

commutes and, of course, $\underline{\overline{\mathcal{C}}} \to \mathbb{1}$ is a morphism in $\mu\nu\mathsf{Poly}_{\mathcal{B}}$;

M2. for any object $\mathcal{D}$ of $\mu\nu\mathsf{Poly}_{\mathcal{A}}$, given a functor $W : \mathbb{1} \to \mathcal{D}$ (which belongs to $\mu\nu\mathsf{Poly}_{\mathcal{A}}$), we have that $\mathrm{dom}^{-1}(\mathcal{D}) \circ W$ is a morphism of $\mu\nu\mathsf{Poly}_{\mathcal{B}}$ such that

$$\begin{tikzcd} \mathbb{1} \arrow[rrr, "W"] \arrow[d, "\mathrm{dom}^{-1}(\mathbb{1})"'] & & & \mathcal{D} \arrow[d, "\mathrm{dom}^{-1}(\mathcal{D})"] \\ \mathbb{1} \arrow[rrr, "\mathrm{dom}^{-1}(\mathcal{D}) \circ W"'] & & & \underline{\overline{\mathcal{D}}} \end{tikzcd} \tag{7.14}$$

commutes;

M3. consider the binary product $\times : \mathcal{A} \times \mathcal{A} \to \mathcal{A}$ (which exists, since G is monadic). We have that $\times : \mathcal{B} \times \mathcal{B} \to \mathcal{B}$ (which is a morphism of $\mu\nu\mathsf{Poly}_{\mathcal{B}}$) is such that we have an isomorphism

$$\begin{tikzcd} \mathcal{A} \times \mathcal{A} \arrow[r, "\times"] \arrow[d, "\mathrm{dom}^{-1}(\mathcal{A} \times \mathcal{A})"'] \arrow[dr, phantom, "\overset{\cong}{\Leftrightarrow}"] & \mathcal{A} \arrow[d, "\mathrm{dom}^{-1}(\mathcal{A})"] \\ \mathcal{B} \times \mathcal{B} \arrow[r, "\times"'] & \mathcal{B} \end{tikzcd} \tag{7.15}$$

since $G : \mathcal{A} \to \mathcal{B}$ preserves products and

$$\mathrm{dom}^{-1}(\mathcal{A}) = G, \qquad \mathrm{dom}^{-1}(\mathcal{A} \times \mathcal{A}) = G \times G;$$

M4. consider the binary coproduct $\sqcup : \mathcal{A} \times \mathcal{A} \to \mathcal{A}$ (which exists, since G is comonadic). We have that $\sqcup : \mathcal{B} \times \mathcal{B} \to \mathcal{B}$ (which is a morphism of

$\mu\nu\mathsf{Poly}_{\mathcal{B}}$) is such that we have an isomorphism

$$\begin{tikzcd} \mathcal{A}\times\mathcal{A} \arrow[r, "\sqcup"] \arrow[d, "G\times G = \mathrm{dom}^{-1}(\mathcal{A}\times\mathcal{A})"'] \arrow[dr, phantom, "\overset{\cong}{\Longleftrightarrow}"] & \mathcal{A} \arrow[d, "G = \mathrm{dom}^{-1}(\mathcal{A})"] \\ \mathcal{B}\times\mathcal{B} \arrow[r, "\sqcup"'] & \mathcal{B} \end{tikzcd} \tag{7.16}$$

since $G : \mathcal{A} \to \mathcal{B}$ preserves coproducts.

M5. for any pair of objects $(\mathcal{C}, \mathcal{D}) \in \mu\nu\mathsf{Poly}_{\mathcal{A}} \times \mu\nu\mathsf{Poly}_{\mathcal{A}}$, we have, of course, that

$$\begin{tikzcd} \mathcal{C}\times\mathcal{D} \arrow[r, "\pi_1"] \arrow[d, "\mathrm{dom}^{-1}(\mathcal{C}\times\mathcal{D})"'] & \mathcal{C} \arrow[d, "\mathrm{dom}^{-1}(\mathcal{C})"] \\ \underline{\overline{\mathcal{C}}}\times\underline{\overline{\mathcal{D}}} \arrow[r, "\pi_1"'] & \underline{\overline{\mathcal{C}}} \end{tikzcd} \qquad \begin{tikzcd} \mathcal{C}\times\mathcal{D} \arrow[r, "\pi_2"] \arrow[d, "\mathrm{dom}^{-1}(\mathcal{C}\times\mathcal{D})"'] & \mathcal{D} \arrow[d, "\mathrm{dom}^{-1}(\mathcal{D})"] \\ \underline{\overline{\mathcal{C}}}\times\underline{\overline{\mathcal{D}}} \arrow[r, "\pi_2"'] & \underline{\overline{\mathcal{D}}} \end{tikzcd} \tag{7.17}$$

commute and $\pi_1 : \underline{\overline{\mathcal{C}}} \times \underline{\overline{\mathcal{D}}} \to \underline{\overline{\mathcal{C}}}$ and $\pi_2 : \underline{\overline{\mathcal{C}}} \times \underline{\overline{\mathcal{D}}} \to \underline{\overline{\mathcal{D}}}$ are morphisms in $\mu\nu\mathsf{Poly}_{\mathcal{B}}$.

M6. given objects $\mathcal{D}', \mathcal{D}'', \mathcal{D}'''$ of $\mu\nu\mathsf{Poly}_{\mathcal{A}}$, if $E : \mathcal{D}' \to \mathcal{D}''$ and $J : \mathcal{D}' \to \mathcal{D}'''$ are morphisms of $\mu\nu\mathsf{Poly}_{\mathcal{A}}$ such that we have natural isomorphisms

$$\begin{aligned} \gamma : \ \underline{\overline{E}} \circ \mathrm{dom}^{-1}(\mathcal{D}') &\cong \mathrm{dom}^{-1}(\mathcal{D}'') \circ E \\ \gamma' : \ \underline{\overline{J}} \circ \mathrm{dom}^{-1}(\mathcal{D}') &\cong \mathrm{dom}^{-1}(\mathcal{D}''') \circ J \end{aligned}$$

in which $\underline{\overline{J}}$ and $\underline{\overline{E}}$ are morphisms of $\mu\nu\mathsf{Poly}_{\mathcal{B}}$, then $\left(\underline{\overline{E}}, \underline{\overline{J}}\right)$ is a morphism in $\mu\nu\mathsf{Poly}_{\mathcal{B}}$ and (γ, γ') defines an isomorphism

$$\left(\underline{\overline{E}}, \underline{\overline{J}}\right) \circ \mathrm{dom}^{-1}(\mathcal{D}') \cong \mathrm{dom}^{-1}(\mathcal{D}'' \times \mathcal{D}''') \circ (E, J). \tag{7.18}$$

M7. if $\mathcal{C}$ is an object of $\mu\nu\mathsf{Poly}_{\mathcal{A}}$ and $H : \mathcal{C} \times \mathcal{A} \to \mathcal{A}$ is a morphism of $\mu\nu\mathsf{Poly}_{\mathcal{A}}$ such that there is an isomorphism

$$\gamma : \underline{\overline{H}} \circ \mathrm{dom}^{-1}(\mathcal{C} \times \mathcal{A}) \cong \mathrm{dom}^{-1}(\mathcal{A}) \circ H$$

in which $\underline{\overline{H}}$ is a morphism of $\mu\nu\mathsf{Poly}_{\mathcal{B}}$, then, since G creates initial algebras and terminal coalgebras, we get that there are natural transformations

$$\begin{aligned} \mu\underline{\overline{H}} \circ \mathrm{dom}^{-1}(\mathcal{A}) &\cong \mathrm{dom}^{-1}(\mathcal{A}) \circ \mu H \\ \nu\underline{\overline{H}} \circ \mathrm{dom}^{-1}(\mathcal{A}) &\cong \mathrm{dom}^{-1}(\mathcal{A}) \circ \nu H \end{aligned}$$

and, of course, $\mu\underline{\overline{H}}$ and $\nu\underline{\overline{H}}$ are morphisms of $\mu\nu\mathsf{Poly}_{\mathcal{B}}$.

■

Corollary 7.10. *Let $\mathcal{C}$ and $\mathcal{D}$ be categories with $\mu\nu$-polynomials. If $G : \mathcal{C} \to \mathcal{D}$ has a left adjoint, then $\mathcal{D} \downarrow G$ has $\mu\nu$-polynomials and*

$$\mathsf{L} : \mathcal{D} \downarrow G \to \mathcal{D} \times \mathcal{C} \tag{7.19}$$

preserves (in fact, creates) $\mu\nu$-polynomials.

Proof: By Corollary 7.2, we have that L is monadic and comonadic (since it has finite products and finite coproducts). 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, $\mathcal{D} \times \mathcal{C}$ has $\mu\nu$-polynomials (constructed pointwise) provided that $\mathcal{D}$ and $\mathcal{C}$ have $\mu\nu$-polynomials. ∎

7.5. The projection $\mathcal{D} \downarrow G \to \mathcal{C}$. Let $\mathcal{C}$ and $\mathcal{D}$ be bi-Cartesian closed categories with finite limits. Recall that $\pi_{\mathcal{C}} : \mathcal{D} \times \mathcal{C} \to \mathcal{C}$ has left and right adjoints. They are respectively given by $W \mapsto (W, \mathbb{0})$ and $W \mapsto (W, \mathbb{1})$. Therefore, assuming that $G : \mathcal{C} \to \mathcal{D}$ has a left adjoint, we get that

$$\mathcal{D} \downarrow G \xrightarrow{\mathsf{L}} \mathcal{D} \times \mathcal{C} \xrightarrow{\pi_2} \mathcal{C} \tag{7.20}$$

has a left adjoint and a right adjoint. Therefore it preserves limits, colimits, initial algebras and terminal coalgebras. Finally, (7.20) preserves the closed structure by Corollary (7.6).

Corollary 7.11. *Let $\mathcal{C}$ and $\mathcal{D}$ be finitely complete bi-Cartesian closed categories that have $\mu\nu$-polynomials. Assume that $G : \mathcal{C} \to \mathcal{D}$ has a left adjoint. In this case, $\mathcal{D} \downarrow G$ is a finitely complete bi-Cartesian closed category with $\mu\nu$-polynomials, and the functor*

$$\mathcal{D} \downarrow G \xrightarrow{\mathsf{L}} \mathcal{D} \times \mathcal{C} \xrightarrow{\pi_2} \mathcal{C}$$

is a bi-Cartesian closed functor that preserves $\mu\nu$-polynomials.

8. Correctness of CHAD, by logical relations

Recall that, by Corollary 4.1, $\Sigma_{\mathbf{CSyn}}\mathbf{LSyn}$ and $\Sigma_{\mathbf{CSyn}}\mathbf{LSyn}^{op}$ are both bi-Cartesian closed categories with $\mu\nu$-polynomials and, hence, we get the following by the universal property of **Syn** (by Corollary 2.1).

Henceforth, following the terminology of Section 6, for any given set X and any vector space V, we denote by

$$\underline{V} : X \to \mathbf{Vect}$$

the family (object of $\mathbf{Fam}(\mathbf{Vect})$ or object of $\mathbf{Fam}(\mathbf{Vect}^{\mathrm{op}})$) constantly equal to V. Moreover, whenever $f : \mathbb{R}^{n_1}\times\cdots\times\mathbb{R}^{n_k} \to \mathbb{R}^m$ is smooth, we respectively denote by

$$Df : \overline{\underline{\mathbb{R}}}^{n_1,\ldots,n_k} \to \underline{\mathbb{R}}^m, \qquad\qquad Df^t : \underline{\mathbb{R}}^m \to \overline{\underline{\mathbb{R}}}^{n_1,\ldots,n_k}$$

the (semantic) derivative and the transpose (semantic) derivative where we denote $\overline{\underline{\mathbb{R}}}^{n_1,\ldots,n_k} = \mathbf{FVect}\,(\pi_1)\,(\underline{\mathbb{R}}^{n_1}) + \cdots + \mathbf{FVect}\,(\pi_k)\,(\underline{\mathbb{R}}^{n_k})$, that is to say,

$$\overline{\underline{\mathbb{R}}}^{n_1,\ldots,n_k} = \underline{\mathbb{R}}^{n_1\cdots n_k} : \mathbb{R}^{n_1}\times\cdots\times\mathbb{R}^{n_k} \to \mathbf{Vect}.$$

Recall that, by Corollary 4.1, we have unique structure-preserving functors $\overrightarrow{\mathcal{D}}(-) : \mathbf{Syn} \to \Sigma_{\mathbf{CSyn}}\mathbf{LSyn}$ and $\overleftarrow{\mathcal{D}}(-) : \mathbf{Syn} \to \Sigma_{\mathbf{CSyn}}\mathbf{LSyn}^{op}$ implementing source-code transformations for forward and reverse AD, respectively. Further, by Corollary 6.9, we have canonical functors

$$\Sigma_{[\![-]\!]}[\![-]\!] : \Sigma_{\mathbf{CSyn}}\mathbf{LSyn} \to \mathbf{Fam}(\mathbf{Vect}), \qquad \Sigma_{[\![-]\!]}[\![-]\!]^{op} : \Sigma_{\mathbf{CSyn}}\mathbf{LSyn}^{op} \to \mathbf{Fam}(\mathbf{Vect}^{\mathrm{op}}).$$

giving a concrete denotational semantics for the entire target language of the forward AD and the reverse AD respectively such that $\Sigma_{[\![-]\!]}[\![-]\!]$ and $\Sigma_{[\![-]\!]}[\![-]\!]^{op}$ are bi-Cartesian closed functors that preserve $\mu\nu$-polynomials. By the definition of the Grothendieck constructions and the fact that

$$\Sigma_{[\![-]\!]}[\![-]\!]\circ\overrightarrow{\mathcal{D}}(-) : \mathbf{Syn} \to \mathbf{Fam}(\mathbf{Vect}), \qquad \Sigma_{[\![-]\!]}[\![-]\!]^{op}\circ\overleftarrow{\mathcal{D}}(-) : \mathbf{Syn} \to \mathbf{Fam}(\mathbf{Vect}^{\mathrm{op}})$$

preserve the Cartesian structure, we have that, for each morphism

$$t : \mathbf{real}^{n_1}\times\cdots\times\mathbf{real}^{n_k} \to \mathbf{real}^m$$

of $\mathbf{Syn}$,

$$\begin{aligned}
&\Sigma_{[\![-]\!]}[\![-]\!]\circ\overrightarrow{\mathcal{D}}(t)\\
&= \left([\![\overrightarrow{\mathcal{D}}(t)]\!]_1 : \mathbb{R}^{n_1}\times\cdots\times\mathbb{R}^{n_k} \to \mathbb{R}^m, [\![\overrightarrow{\mathcal{D}}(t)]\!]_2 : \overline{\underline{\mathbb{R}}}^{n_1,\ldots,n_k} \to \mathbf{FVect}\left([\![\overrightarrow{\mathcal{D}}(t)]\!]_1\right)(\underline{\mathbb{R}}^m)\right)\\
&= \left([\![\overrightarrow{\mathcal{D}}(t)]\!]_1 : \mathbb{R}^{n_1}\times\cdots\times\mathbb{R}^{n_k} \to \mathbb{R}^m, [\![\overrightarrow{\mathcal{D}}(t)]\!]_2 : \overline{\underline{\mathbb{R}}}^{n_1,\ldots,n_k} \to \underline{\mathbb{R}}^m\right)
\end{aligned}$$

$$\begin{aligned}
&\Sigma_{[\![-]\!]}[\![-]\!]^{\mathrm{op}}\circ\overleftarrow{\mathcal{D}}(t)\\
&= \left([\![\overleftarrow{\mathcal{D}}(t)]\!]_1 : \mathbb{R}^{n_1}\times\cdots\times\mathbb{R}^{n_k} \to \mathbb{R}^m, [\![\overleftarrow{\mathcal{D}}(t)]\!]_2 : \mathbf{FVect}\left([\![\overleftarrow{\mathcal{D}}(t)]\!]_1\right)(\underline{\mathbb{R}}^m) \to \overline{\underline{\mathbb{R}}}^{n_1,\ldots,n_k}\right)\\
&= \left([\![\overleftarrow{\mathcal{D}}(t)]\!]_1 : \mathbb{R}^{n_1}\times\cdots\times\mathbb{R}^{n_k} \to \mathbb{R}^m, [\![\overleftarrow{\mathcal{D}}(t)]\!]_2 : \underline{\mathbb{R}}^m \to \overline{\underline{\mathbb{R}}}^{n_1,\ldots,n_k}\right)
\end{aligned}$$

for some morphism $\left([\![\overrightarrow{\mathcal{D}}(t)]\!]_1, [\![\overleftarrow{\mathcal{D}}(t)]\!]_1\right)$ of $\mathbf{Set}\times\mathbf{Set}$ and some morphism $\left([\![\overrightarrow{\mathcal{D}}(t)]\!]_2, [\![\overleftarrow{\mathcal{D}}(t)]\!]_2\right)$ of $\mathbf{FVect}\,(\underline{\mathbb{R}}^{n_1}\times\cdots\times\underline{\mathbb{R}}^{n_k})\times\mathbf{FVect}\,(\underline{\mathbb{R}}^{n_1}\times\cdots\times\underline{\mathbb{R}}^{n_k})$. In this section, we prove that, for any such morphism

$$t : \mathbf{real}^{n_1}\times\cdots\times\mathbf{real}^{n_k} \to \mathbf{real}^m$$

of $\mathbf{Syn}$, we have that $[\![t]\!]$ is smooth and

$$[\![\overrightarrow{\mathcal{D}}(t)]\!]_1 = [\![\overleftarrow{\mathcal{D}}(t)]\!]_1 = [\![t]\!], \qquad [\![\overrightarrow{\mathcal{D}}(t)]\!]_2 = D[\![t]\!], \qquad [\![\overleftarrow{\mathcal{D}}(t)]\!]_2 = D[\![t]\!]^t.$$

That is to say, in this section, we prove:

Theorem 8.1 (Correctness of CHAD)**.** *For any well-typed program*

$$x_1 : \mathbf{real}^{n_1}, \ldots, x_k : \mathbf{real}^{n_k} \vdash t : \mathbf{real}^m$$

we have that

$$[\![\overrightarrow{\mathcal{D}}(t)]\!]_1 = [\![\overleftarrow{\mathcal{D}}(t)]\!]_1 = [\![t]\!], \qquad [\![\overrightarrow{\mathcal{D}}(t)]\!]_2 = D[\![t]\!], \qquad [\![\overleftarrow{\mathcal{D}}(t)]\!]_2 = D[\![t]\!]^t.$$

Note that t might, in particular, have subprograms that use higher-order functions, sum types and (co)inductive types. In order to prove the result above, we construct the scones w.r.t. the following functors

$$\begin{array}{ll} \overrightarrow{G} : \mathbf{Set} \times \mathbf{Fam}(\mathbf{Vect}) \to \mathbf{Set}, & \overleftarrow{G} : \mathbf{Set} \times \mathbf{Fam}(\mathbf{Vect}^{\mathrm{op}}) \to \mathbf{Set}, \\ \overrightarrow{G} = \mathbf{Set} \times \mathbf{Fam}(\mathbf{Vect})\left((\mathbb{R}, (\mathbb{R}, \underline{\mathbb{R}})), -\right), & \overleftarrow{G} = \mathbf{Set} \times \mathbf{Fam}(\mathbf{Vect}^{\mathrm{op}})\left((\mathbb{R}, (\mathbb{R}, \underline{\mathbb{R}})), -\right). \end{array}$$

Since **Set**, **Fam**(**Vect**) and **Fam**($\mathbf{Vect}^{\mathrm{op}}$) have coproducts, we get that, for each $W \in \mathbf{Set}$, the copowers

$$\begin{array}{rcl} W \otimes (\mathbb{R}, (\mathbb{R}, \underline{\mathbb{R}})) & \cong & \coprod_{x \in W} (\mathbb{R}, (\mathbb{R}, \underline{\mathbb{R}})) \\ & \cong & (W \otimes \mathbb{R}, (W \otimes \mathbb{R}, \underline{\mathbb{R}} : W \otimes \mathbb{R} \to \mathbf{Vect})) \end{array}$$

exist in $\mathbf{Set} \times \mathbf{Fam}(\mathbf{Vect})$ and $\mathbf{Set} \times \mathbf{Fam}(\mathbf{Vect}^{\mathrm{op}})$. Hence the functors $W \mapsto W \otimes (\mathbb{R}, (\mathbb{R}, \underline{\mathbb{R}}))$ give the left adjoints to $\overrightarrow{G}$ and $\overleftarrow{G}$ respectively.

Moreover, since **Set**, **Fam**(**Vect**) and **Fam**($\mathbf{Vect}^{\mathrm{op}}$) are Cartesian closed locally presentable categories (see 6.1 and 6.2), we have that $\mathbf{Set} \times \mathbf{Fam}(\mathbf{Vect})$ and $\mathbf{Set} \times \mathbf{Fam}(\mathbf{Vect}^{\mathrm{op}})$ are Cartesian closed locally presentable categories as well. In particular, $\mathbf{Set} \times \mathbf{Fam}(\mathbf{Vect})$ and $\mathbf{Set} \times \mathbf{Fam}(\mathbf{Vect}^{\mathrm{op}})$ are finitely complete Cartesian closed categories that have $\mu\nu$-polynomials (see Proposition 6.3). Therefore, by Corollary 7.11:

Lemma 8.1.

$$\overrightarrow{\mathbf{Scone}} := \mathcal{D} \downarrow \overrightarrow{G}, \qquad \overleftarrow{\mathbf{Scone}} := \mathcal{D} \downarrow \overleftarrow{G} \tag{8.1}$$

are finitely complete bi-Cartesian closed categories with $\mu\nu$-polynomials and the projections

$$\overrightarrow{\pi} : \overrightarrow{\mathbf{Scone}} \to \mathbf{Fam}(\mathbf{Vect}) \times \mathbf{Set}, \qquad \overleftarrow{\pi} : \overleftarrow{\mathbf{Scone}} \to \mathbf{Fam}(\mathbf{Vect}^{\mathrm{op}}) \times \mathbf{Set} \tag{8.2}$$

are bi-Cartesian closed functors and preserve $\mu\nu$-polynomials.

Lemma 8.2. *There are unique functors*

$$\overrightarrow{\llbracket - \rrbracket} : \mathbf{Syn} \to \overrightarrow{\mathbf{Scone}}, \qquad \overleftarrow{\llbracket - \rrbracket} : \mathbf{Syn} \to \overleftarrow{\mathbf{Scone}} \tag{8.3}$$

that preserve the bi-Cartesian closed structure and $\mu\nu$-polynomials such that

- *for each n-dimensional array* $\mathbf{real}^n \in \mathbf{Syn}$,

$$\overrightarrow{\llbracket \mathbf{real}^n \rrbracket} = \left(\left\{ (f,(g,h)) \in \overrightarrow{G}(\mathbb{R}^n, (\mathbb{R}^n, \underline{\mathbb{R}}^n)) : \, f \textit{ is smooth, } f = g \textit{ and } h = Df \right\}, (\mathbb{R}^n, (\mathbb{R}^n, \underline{\mathbb{R}}^n)) \right),$$

  with the obvious inclusion in $\overrightarrow{G}(\mathbb{R}^n, (\mathbb{R}^n, \underline{\mathbb{R}}^n))$, and

$$\overleftarrow{\llbracket \mathbf{real}^n \rrbracket} = \left(\left\{ (f,(g,h)) \in \overleftarrow{G}(\mathbb{R}^n, (\mathbb{R}^n, \underline{\mathbb{R}}^n)) : \, f \textit{ is smooth, } f = g \textit{ and } h = Df^t \right\}, (\mathbb{R}^n, (\mathbb{R}^n, \underline{\mathbb{R}}^n)) \right),$$

  with the obvious inclusion in $\overleftarrow{G}(\mathbb{R}^n, (\mathbb{R}^n, \underline{\mathbb{R}}^n))$;
- *for each primitive* $\mathrm{op} \in \mathsf{Op}^m_{n_1,\ldots,n_k}$,

$$\overrightarrow{\llbracket \mathrm{op} \rrbracket} = \left(\underrightarrow{\llbracket \tilde{\mathrm{op}} \rrbracket}, \left(\llbracket \mathrm{op} \rrbracket, \llbracket \overrightarrow{\mathcal{D}}(\mathrm{op}) \rrbracket \right) \right), \qquad \overleftarrow{\llbracket \mathrm{op} \rrbracket} = \left(\underleftarrow{\llbracket \tilde{\mathrm{op}} \rrbracket}, \left(\llbracket \mathrm{op} \rrbracket, \llbracket \overleftarrow{\mathcal{D}}(\mathrm{op}) \rrbracket \right) \right).$$

  in which $\underrightarrow{\llbracket \tilde{\mathrm{op}} \rrbracket}$ and $\underleftarrow{\llbracket \tilde{\mathrm{op}} \rrbracket}$ are the unique functions such that the pairs above give morphisms of $\overrightarrow{\mathbf{Scone}}$ and $\overleftarrow{\mathbf{Scone}}$ respectively.

Proof: By Corollary 2.1 (the universal property of the source language **Syn**), it is enough to prove that, indeed, there are functions $\underrightarrow{\llbracket \tilde{\mathrm{op}} \rrbracket}$ and $\underleftarrow{\llbracket \tilde{\mathrm{op}} \rrbracket}$ such that $\left(\underrightarrow{\llbracket \tilde{\mathrm{op}} \rrbracket}, \left(\llbracket \mathrm{op} \rrbracket, \llbracket \overrightarrow{\mathcal{D}}(\mathrm{op}) \rrbracket \right) \right)$ and $\left(\underleftarrow{\llbracket \tilde{\mathrm{op}} \rrbracket}, \left(\llbracket \mathrm{op} \rrbracket, \llbracket \overleftarrow{\mathcal{D}}(\mathrm{op}) \rrbracket \right) \right)$ yield morphisms in $\overrightarrow{\mathbf{Scone}}$ and $\overleftarrow{\mathbf{Scone}}$ respectively. That is to say, it is enough to see the following.

- whenever $(f,(g,h))$ is such that f is a smooth curve $\mathbb{R} \to \mathbb{R}^{n_1} \times \cdots \times \mathbb{R}^{n_k}$, $f = g$ and $h = Df$, we have that, for any primitive $\mathrm{op} \in \mathsf{Op}^m_{n_1,\ldots,n_k}$,
  - $\llbracket \mathrm{op} \rrbracket \circ f$ is smooth;
  - $\mathbf{FVect}\,(g) \left(\llbracket \overrightarrow{\mathcal{D}}(\mathrm{op}) \rrbracket_2 \right) \circ h$ is the derivative of $\llbracket \mathrm{op} \rrbracket \circ f$ by the chain rule, provided that $\llbracket \overrightarrow{\mathcal{D}}(\mathrm{op}) \rrbracket_2$ indeed is transpose derivative of $\llbracket \mathrm{op} \rrbracket$.

  Therefore

$$\begin{aligned}
&\left(\llbracket \mathrm{op} \rrbracket, \llbracket \overrightarrow{\mathcal{D}}(\mathrm{op}) \rrbracket \right) \circ (f,(g,h)) \\
&= \left(\llbracket \mathrm{op} \rrbracket \circ f, \llbracket \overrightarrow{\mathcal{D}}(\mathrm{op}) \rrbracket \circ (g,h) \right) \\
&= \left(\llbracket \mathrm{op} \rrbracket \circ f, \left(\llbracket \overrightarrow{\mathcal{D}}(\mathrm{op}) \rrbracket_1 \circ g, \mathbf{FVect}\,(g) \left(\llbracket \overrightarrow{\mathcal{D}}(\mathrm{op}) \rrbracket_2 \right) \circ h \right) \right) \\
&= \left(\llbracket \mathrm{op} \rrbracket \circ f, \left(\llbracket \mathrm{op} \rrbracket \circ g, \mathbf{FVect}\,(g) \left(\llbracket \overrightarrow{\mathcal{D}}(\mathrm{op}) \rrbracket_2 \right) \circ h \right) \right)
\end{aligned}$$

indeed satisfies the following: $[\![\mathrm{op}]\!] \circ f = [\![\mathrm{op}]\!] \circ g$ and

$$\mathbf{FVect}\,(g)\,\big([\![\overrightarrow{\mathcal{D}}(\mathrm{op})]\!]_2\big) \circ h$$

is the derivative of $[\![\mathrm{op}]\!] \circ g$.

- whenever $(f,(g,h))$ is such that f is a smooth curve $\mathbb{R} \to \mathbb{R}^{n_1} \times \cdots \times \mathbb{R}^{n_k}$, $f = g$ and $h = Df^t$, we have that, for any primitive $\mathrm{op} \in \mathsf{Op}^m_{n_1,\ldots,n_k}$,
  - $[\![\mathrm{op}]\!] \circ f$ is smooth;
  - $h \circ \mathbf{FVect}\,(g)\,\big([\![\overleftarrow{\mathcal{D}}(\mathrm{op})]\!]_2\big)$ is the transpose derivative of $[\![\mathrm{op}]\!] \circ f$ by the chain rule, provided that $[\![\overleftarrow{\mathcal{D}}(\mathrm{op})]\!]_2$ indeed is transpose derivative of $[\![\mathrm{op}]\!]$.

  Therefore

$$\begin{aligned}
&\big([\![\mathrm{op}]\!], [\![\overleftarrow{\mathcal{D}}(\mathrm{op})]\!]\big) \circ (f,(g,h)) \\
&= \big([\![\mathrm{op}]\!] \circ f, [\![\overleftarrow{\mathcal{D}}(\mathrm{op})]\!] \circ (g,h)\big) \\
&= \Big([\![\mathrm{op}]\!] \circ f, \big([\![\overleftarrow{\mathcal{D}}(\mathrm{op})]\!]_1 \circ g, h \circ \mathbf{FVect}\,(g)\,\big([\![\overleftarrow{\mathcal{D}}(\mathrm{op})]\!]_2\big)\big)\Big) \\
&= \Big([\![\mathrm{op}]\!] \circ f, \big([\![\mathrm{op}]\!] \circ g, h \circ \mathbf{FVect}\,(g)\,\big([\![\overleftarrow{\mathcal{D}}(\mathrm{op})]\!]_2\big)\big)\Big)
\end{aligned}$$

  indeed satisfies the following: $[\![\mathrm{op}]\!] \circ f = [\![\mathrm{op}]\!] \circ g$ and

$$h \circ \mathbf{FVect}\,(g)\,\big([\![\overrightarrow{\mathcal{D}}(\mathrm{op})]\!]_2\big)$$

  is the transpose derivative of $[\![\mathrm{op}]\!] \circ g$.

Indeed, this follows from the proof of [40, Lemma 8.1]. ∎

By the definitions of $\overrightarrow{[\![-]\!]}$ and $\overleftarrow{[\![-]\!]}$ above, for each $\mathbf{real}^n \in \mathbf{Syn}$ and any primitive $\mathrm{op} \in \mathsf{Op}^m_{n_1,\ldots,n_k}$,

$$\begin{aligned}
\big([\![\mathbf{real}^n]\!], \Sigma_{[\![-]\!]}[\![-]\!]\,(\overrightarrow{\mathcal{D}}(\mathbf{real}^n))\big) &= (\mathbb{R}^n, (\mathbb{R}^n, \underline{\mathbb{R}}^n)) \\
&= \overrightarrow{\pi}\left(\overrightarrow{[\![\mathbf{real}^n]\!]}\right), \\
\big([\![\mathbf{real}^n]\!], \Sigma_{[\![-]\!]}[\![-]\!]^{\mathrm{op}}\,(\overleftarrow{\mathcal{D}}(\mathbf{real}^n))\big) &= (\mathbb{R}^n, (\mathbb{R}^n, \underline{\mathbb{R}}^n)) \\
&= \overleftarrow{\pi}\left(\overleftarrow{[\![\mathbf{real}^n]\!]}\right),
\end{aligned}$$

$$\begin{aligned}
\big([\![\mathrm{op}]\!], \Sigma_{[\![-]\!]}[\![-]\!]\,(\overrightarrow{\mathcal{D}}(\mathrm{op}))\big) &= ([\![\mathrm{op}]\!], ([\![\mathrm{op}]\!], D[\![\mathrm{op}]\!])) \\
&= \overrightarrow{\pi}\left(\overrightarrow{[\![\mathrm{op}]\!]}\right), \\
\big([\![\mathrm{op}]\!], \Sigma_{[\![-]\!]}[\![-]\!]^{\mathrm{op}}\,(\overleftarrow{\mathcal{D}}(\mathrm{op}))\big) &= \big([\![\mathrm{op}]\!], \big([\![\mathrm{op}]\!], D[\![\mathrm{op}]\!]^t\big)\big)
\end{aligned}$$

$$= \overleftarrow{\pi}\left(\overleftarrow{[\![\mathrm{op}]\!]}\right).$$

Therefore, by the universal property of **Syn** (Corollary 2.1), since

$$\begin{array}{ll} \overrightarrow{\pi} \circ \overrightarrow{[\![-]\!]}, & \left([\![-]\!] \times \Sigma_{[\![-]\!]}[\![-]\!]\right) \circ \left([\![-]\!], \overrightarrow{\mathcal{D}}(-)\right) \\ \overleftarrow{\pi} \circ \overleftarrow{[\![-]\!]}, & \left([\![-]\!] \times \Sigma_{[\![-]\!]}[\![-]\!]^{\mathrm{op}}\right) \circ \left([\![-]\!], \overleftarrow{\mathcal{D}}(-)\right) \end{array}$$

are (strictly) bi-Cartesian closed functors that (strictly) preserve $\mu\nu$-polynomials, we conclude that the diagrams

$$\begin{tikzcd} \mathbf{Syn} \arrow[r, "{([\![-]\!], \overrightarrow{\mathcal{D}}(-))}"] \arrow[d, "\overrightarrow{[\![-]\!]}"'] & \mathbf{Syn} \times \Sigma_{\mathbf{CSyn}}\mathbf{LSyn} \arrow[d, "{[\![-]\!] \times \Sigma_{[\![-]\!]}[\![-]\!]}"] \\ \overrightarrow{\mathbf{Scone}} \arrow[r, "\overrightarrow{\pi}"'] & \mathbf{Set} \times \mathbf{Fam}(\mathbf{Vect}) \end{tikzcd} \qquad \begin{tikzcd} \mathbf{Syn} \arrow[r, "{([\![-]\!], \overleftarrow{\mathcal{D}}(-))}"] \arrow[d, "\overleftarrow{[\![-]\!]}"'] & \mathbf{Syn} \times \Sigma_{\mathbf{CSyn}}\mathbf{LSyn}^{\mathrm{op}} \arrow[d, "{[\![-]\!] \times \Sigma_{[\![-]\!]}[\![-]\!]^{\mathrm{op}}}"] \\ \overleftarrow{\mathbf{Scone}} \arrow[r, "\overleftarrow{\pi}"'] & \mathbf{Set} \times \mathbf{Fam}(\mathbf{Vect}^{\mathrm{op}}) \end{tikzcd}$$

commute. This implies that, for any morphism $t : \mathbf{real}^{n_1} \times \cdots \times \mathbf{real}^{n_k} \to \mathbf{real}^m$ of **Syn**, we have

$$\begin{aligned} \overrightarrow{\pi} \circ \overrightarrow{[\![t]\!]} &= \left([\![t]\!], \Sigma_{[\![-]\!]}[\![-]\!] \circ \overrightarrow{\mathcal{D}}(t)\right) \\ &= \left([\![t]\!], \left([\![\overrightarrow{\mathcal{D}}(t)]\!]_1 : \mathbb{R}^{n_1} \times \cdots \times \mathbb{R}^{n_k} \to \mathbb{R}^m, [\![\overrightarrow{\mathcal{D}}(t)]\!]_2 : \underline{\mathbb{R}}^{n_1,\ldots,n_k} \to \underline{\mathbb{R}}^m\right)\right) \end{aligned}$$

$$\begin{aligned} \overleftarrow{\pi} \circ \overleftarrow{[\![t]\!]} &= \left([\![t]\!], \Sigma_{[\![-]\!]}[\![-]\!]^{\mathrm{op}} \circ \overleftarrow{\mathcal{D}}(t)\right) \\ &= \left([\![t]\!], \left([\![\overleftarrow{\mathcal{D}}(t)]\!]_1 : \mathbb{R}^{n_1} \times \cdots \times \mathbb{R}^{n_k} \to \mathbb{R}^m, [\![\overleftarrow{\mathcal{D}}(t)]\!]_2 : \underline{\mathbb{R}}^m \to \underline{\mathbb{R}}^{n_1,\ldots,n_k}\right)\right) \end{aligned}$$

which show, by the definitions of $\overrightarrow{[\![\mathbf{real}^m]\!]}$ and $\overleftarrow{[\![\mathbf{real}^m]\!]}$ (logical relations), that, for any curve $\gamma : \mathbb{R} \to \mathbb{R}^{n_1} \times \cdots \times \mathbb{R}^{n_k}$,

- $[\![\overrightarrow{\mathcal{D}}(t)]\!]_1 \circ \gamma$ and $[\![\overleftarrow{\mathcal{D}}(t)]\!]_2 \circ \gamma$ are smooth;
- $[\![\overrightarrow{\mathcal{D}}(t)]\!]_1 \circ \gamma = [\![t]\!] \circ \gamma = [\![\overleftarrow{\mathcal{D}}(t)]\!]_1 \circ \gamma$;
- $\mathbf{FVect}\,(\gamma)\left([\![\overrightarrow{\mathcal{D}}(t)]\!]_2\right) \circ D\gamma$ is the derivative of $[\![\overrightarrow{\mathcal{D}}(t)]\!]_1 \circ \gamma$;
- $D\gamma^t \circ \mathbf{FVect}\,(\gamma)\left([\![\overleftarrow{\mathcal{D}}(t)]\!]_2\right)$ is the transpose derivative of $[\![\overleftarrow{\mathcal{D}}(t)]\!]_1 \circ \gamma$.

Of course, the above implies that $[\![\overrightarrow{\mathcal{D}}(t)]\!]_1 = [\![t]\!] = [\![\overleftarrow{\mathcal{D}}(t)]\!]_1$. But, also, by Boman's theorem [8], we conclude that the above implies that $[\![t]\!]$ is smooth, $[\![\overrightarrow{\mathcal{D}}(t)]\!]_2$ is the derivative of $[\![\overrightarrow{\mathcal{D}}(t)]\!]_1 = [\![t]\!]$, and $[\![\overleftarrow{\mathcal{D}}(t)]\!]_2$ is the transpose derivative of $[\![\overleftarrow{\mathcal{D}}(t)]\!]_1 = [\![t]\!]^*$. This completes the proof of Theorem 8.1.

*See the argument of [40, Theorem 8.3]

9. 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.

9.1. Addressing expression blow-up and sharing common subcomputations. We can observe that our source-code transformations of Appendix H 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 §5, 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 $\overrightarrow{\mathcal{D}}(-) : \mathbf{Syn} \to \Sigma_{\mathbf{CSyn}}\mathbf{LSyn}$ and $\overleftarrow{\mathcal{D}}(-) : \mathbf{Syn} \to \Sigma_{\mathbf{CSyn}}\mathbf{LSyn}^{op}$ in the sense that $\overrightarrow{\mathcal{D}}_{\overline{\Gamma}}(t) \overset{\beta\eta+}{=} \langle \overrightarrow{\mathcal{D}}_{\overline{\Gamma}}(t)_1, \underline{\lambda}\mathsf{v}.\overrightarrow{\mathcal{D}}_{\overline{\Gamma}}(t)_2\rangle$ and $\overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(t) \overset{\beta\eta+}{=} \langle \overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(t)_1, \underline{\lambda}\mathsf{v}.\overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(t)_2\rangle$ and hence are equivalent to the inficient 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 $\overrightarrow{\mathcal{D}}_{\overline{\Gamma}}(C[t_1, \ldots, t_n])$ (resp. $\overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(C[t_1, \ldots, t_n])$) of a term former $C[t_1, \ldots, t_n]$ that takes n arguments t_1, $\ldots$, 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 necesssary condition if this AD technique is to scale up to large code-bases.

Corollary 9.1 (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 run-time complexity of the generated code remains to be studied.

9.2. 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 (non-trivial) 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:

$$
\begin{array}{lll@{\qquad}lll}
\underline{\alpha}^\dagger & \stackrel{\text{def}}{=} & \underline{\alpha} & (\mathbf{case}\, t\, \mathbf{of}\, \{\ell_1 x_1 \to \underline{\tau}_1 \mid \cdots \ell_n x_n \to \underline{\tau}_n\})^\dagger & \stackrel{\text{def}}{=} & \underline{\tau}_1{}^\dagger \vee \cdots \vee \underline{\tau}_n{}^\dagger \\
\underline{\mathbf{real}}^{n\,\dagger} & \stackrel{\text{def}}{=} & \underline{\mathbf{real}}^n & (\underline{\mu}\underline{\alpha}.\underline{\tau})^\dagger & \stackrel{\text{def}}{=} & \underline{\mu}\underline{\alpha}.\underline{\tau}^\dagger \\
\underline{\mathbf{1}}^\dagger & \stackrel{\text{def}}{=} & \underline{\mathbf{1}} & (\underline{\nu}\underline{\alpha}.\underline{\tau})^\dagger & \stackrel{\text{def}}{=} & \underline{\mu}\underline{\alpha}.\underline{\tau}^\dagger \\
(\underline{\tau} \ast \underline{\sigma})^\dagger & \stackrel{\text{def}}{=} & \underline{\tau}^\dagger \ast \underline{\sigma}^\dagger & (\underline{\tau} \multimap \underline{\sigma})^\dagger & \stackrel{\text{def}}{=} & \underline{\tau}^\dagger \multimap \underline{\sigma}^\dagger \\
(\Pi x : \tau.\underline{\sigma})^\dagger & \stackrel{\text{def}}{=} & \Pi x : \tau^\dagger.\underline{\sigma}^\dagger & (\Pi x : \tau.\sigma)^\dagger & \stackrel{\text{def}}{=} & \Pi x : \tau^\dagger.\sigma^\dagger \\
(\Sigma x : \tau.\underline{\sigma})^\dagger & \stackrel{\text{def}}{=} & \Sigma x : \tau^\dagger.\underline{\sigma}^\dagger & (\Sigma x : \tau.\sigma)^\dagger & \stackrel{\text{def}}{=} & \Sigma x : \tau^\dagger.\sigma^\dagger.
\end{array}
$$

In fact, seeing that $(\mathbf{case}\, t\, \mathbf{of}\, \{\ell_1 x_1 \to \underline{\tau}_1 \mid \cdots \ell_n x_n \to \underline{\tau}_n\})$-types were the only source of type dependency in our language while these are translated to non-depndent types, all Π- and Σ-types are simply translated to powers, copowers, function types and product types:

$$
\begin{array}{lll@{\qquad}lll}
(\Pi x : \tau.\underline{\sigma})^\dagger & = & \tau^\dagger \to \underline{\sigma}^\dagger & (\Pi x : \tau.\sigma)^\dagger & = & \tau^\dagger \to \sigma^\dagger \\
(\Sigma x : \tau.\underline{\sigma})^\dagger & = & !\tau^\dagger \otimes \underline{\sigma}^\dagger & (\Sigma x : \tau.\sigma)^\dagger & = & \tau^\dagger \ast \sigma^\dagger.
\end{array}
$$

Our translation $(-)^\dagger$ is the identity on programs.

The types $\underline{\tau}_1 \vee \cdots \vee \underline{\tau}_n$ 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.

9.3. Removing linear types from the target language.

Basics. As discussed in detail in [40, 39] and demonstrated in the Haskell implementation available at `https://github.com/VMatthijs/CHAD`, the types $\underline{\mathbf{real}}^n$, $\underline{\mathbf{1}}$, $\underline{\tau}*\underline{\sigma}$, $\tau \to \underline{\sigma}$, $!\tau \otimes \underline{\sigma}$ and $\underline{\tau} \multimap \underline{\sigma}$ (and, obviously, the ordinary Cartesian function and product types $\tau \to \sigma$ and $\tau*\sigma$) together with their terms can all be implemented in a standard functional language. The core idea is to implement $\underline{\tau}$ as the type $\underline{\tau}^\ddagger$:

$$
\begin{array}{lcllcl}
\underline{\mathbf{real}}^{n\ddagger} & \stackrel{\text{def}}{=} & \mathbf{real}^n & (\tau \to \underline{\sigma})^\ddagger & \stackrel{\text{def}}{=} & \tau^\ddagger \to \underline{\sigma}^\ddagger \\
\underline{\mathbf{1}}^\ddagger & \stackrel{\text{def}}{=} & \mathbf{1} & (!\tau \otimes \underline{\sigma})^\ddagger & \stackrel{\text{def}}{=} & [(\tau^\ddagger, \underline{\sigma}^\ddagger)] \\
(\underline{\tau}*\underline{\sigma})^\ddagger & \stackrel{\text{def}}{=} & \underline{\tau}^\ddagger*\underline{\sigma}^\ddagger & (\underline{\tau} \multimap \underline{\sigma})^\ddagger & \stackrel{\text{def}}{=} & \underline{\tau}^\ddagger \to \underline{\sigma}^\ddagger.
\end{array}
$$

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 $\underline{\tau}^\ddagger \to \underline{\sigma}^\ddagger$. As discussed in [40, 39] 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 $\underline{\tau}_1 \vee \cdots \vee \underline{\tau}_n$, $\underline{\mu\alpha}.\underline{\tau}$ and $\underline{\nu\alpha}.\underline{\tau}$ required to perform AD on source languages that additionally use sum types, inductive types and coinductive types.

Linear sum types $\underline{\tau}_1 \vee \cdots \vee \underline{\tau}_n$. We briefly outline three possible implementations $(\underline{\tau}_1 \vee \cdots \vee \underline{\tau}_n)^\ddagger$ of the linear sum types $\underline{\tau}_1 \vee \cdots \vee \underline{\tau}_n$:

(1) as a finite (bi)product $\underline{\tau}_1{}^\ddagger * \cdots * \underline{\tau}_n{}^\ddagger$;
(2) as a finite lifted sum $\{\mathit{Zero} \mid \mathit{Opt}_1\, \underline{\tau}_1{}^\ddagger \mid \cdots \mid \mathit{Opt}_n\, \underline{\tau}_n{}^\ddagger\}$;
(3) as a finite sum $\{\mathit{Opt}_1\, \underline{\tau}_1{}^\ddagger \mid \cdots \mid \mathit{Opt}_n\, \underline{\tau}_n{}^\ddagger\}$.

Approach 1 has the advantage that we can keep the implementation total. As demonstrated in Appendix I, this allows us the easily extend the logical relations argument for the correctness of the applied implementation of

[40, 39] (in actual Haskell, available at `https://github.com/VMatthijs/CHAD`). Categorically, what is going on is that, for a locally indexed category $\mathcal{L} : \mathcal{C}^{op} \to \mathbf{Cat}$, with indexed finite biproducts, $(X_1 \times \cdots \times X_n, A_1 \times \cdots \times A_n)$ is a weak product of (X_1, A_1), ..., (X_n, A_n) in both $\Sigma_{\mathcal{C}}\mathcal{L}$ and $\Sigma_{\mathcal{C}}\mathcal{L}^{op}$: i.e. a product for which the η-law may fail. The logical relations proof of Appendix I lifts these weak biproducts 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 non-zero component, corresponding to connected component the current primal is in. We pay for this more efficient representation in two ways:

- addition on the (co)tangent space is defined by

$$Zero + x = x \qquad x + Zero = x \qquad Opt_i(t) + Opt_i(s) = Opt_i(t + s)$$

  and hence is a partial operation that throws an error if we try to add $Opt_i(t) + Opt_j(s)$ for $i \neq j$; in particular, the nice theoretical story about **CMon**-enriched fibres breaks down;
- 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 trigerred in practice. However, it complicates a direct correctness proof of this approach as we need to work with a semantics with partial functions. This is the approach of 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 unneccesary 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 $\underline{0} : \overrightarrow{\mathcal{D}}(\tau)_1 \to \overrightarrow{\mathcal{D}}(\tau)_2$ and $\underline{0} : \overleftarrow{\mathcal{D}}(\tau)_1 \to \overleftarrow{\mathcal{D}}(\tau)_2$, 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.

Linear inductive and coinductive types $\underline{\mu}\underline{\alpha}.\underline{\tau}$ and $\underline{\nu}\underline{\alpha}.\underline{\tau}$. 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 $F : \mathbf{Vect} \to \mathbf{Vect}$ are of the form $W \mapsto L(A)+W^n$, where $L \dashv U : \mathbf{Set} \to \mathbf{Vect}$ is the usual free-forgetful adjunction. Therefore, $U \circ F = H \circ U$ for the polynomial $H : \mathbf{Set} \to \mathbf{Set}$ defined by $S \mapsto U(L(A)) \times S^n$. As the forgetful functor $F : \mathbf{Vect} \to \mathbf{Set}$ is monadic, it creates terminal coalgebras, hence hence $U(\nu F) = \nu H$. This suggests that we might be able to implement $(\underline{\nu}\underline{\alpha}.\underline{\tau})^{\ddagger}$ as the plain coinductive type $\nu\alpha.\underline{\tau}^{\ddagger}$, where $\underline{\alpha}^{\ddagger} \stackrel{\mathrm{def}}{=} \alpha$.

Similarly, we have that $F \circ L = L \circ E$ for the polynomial $E : \mathbf{Set} \to \mathbf{Set}$ defined by $E(X) = A \sqcup \bigsqcup_n X$. Therefore, we have that

$$\mu F = L(\mu E) = (\mu E) \to \mathbb{R}.$$

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.

10. 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 [43] and variants of reverse mode AD seemingly being reinvented several times, for example, by [25, 37]. For brief reviews of this complex history and the basic ideas behind AD, we refer the reader to [7]. For a more comprehensive account of the traditional work on AD, see the standard reference text [16].

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.

[33] 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 an 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. [4] recently simplified this code transformation and formalized its correctness.

Prompted by [35], there has, more recently, been a push in the programming language community to learn from [33] 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, [42] 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 a formalizations and correctness proofs of reverse AD on expressive languages through the use of custom operational semantics. [1] 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. [31] 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, [9, 32] give an analysis of reverse AD on a simply typed λ-calculus and PCF. Notably, [9] 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, [22] 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 [6, 18]) as well as an asymptotic complexity proof about the full code transformation plus evaluator.

[13] 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 [9, 32]: 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. [41, 39] 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 [41] relates CHAD to the approach of [33, 4], [39] and its extended version [40] give a (denotational) semantic foundation and correctness proof for CHAD, using a combination of logical relations techniques that [6, 18, 17] 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 [14, 11]. 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.

Appendix A. Pseudo-preterminal objects in Cat

The appropriate 2-dimensional analogous to preterminal objects are the pseudo-preterminal ones. Namely, in the case of **Cat**, an object W in **Cat** is *pseudo-preterminal* if the category of functors $\mathbf{Cat}[X, W]$ is a groupoid for any X in **Cat**.

Lemma A.1 establishes that the initial and terminal categories are, up to equivalence, the only pseudo-preterminal objects of **Cat**.

Lemma A.1 (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 $W \to \mathbb{1}$ is an equivalence.*
- *ii The projection $\pi_W : W \times W \to W$ is an equivalence.*
- *iii The identity $\mathrm{id}_W : W \to W$ is naturally isomorphic to a constant functor $c : W \to W$.*
- *iv If $f, g : X \to W$ are functors, then there is a natural isomorphism $f \cong g$ (that is to say, W is pseudo-preterminal).*

Proof: Assuming (i), denoting by $t : W \to \mathbb{1}$ the unique functor, we have that π_W is the composition $W \times W \xrightarrow{\mathrm{id}_W \times t} W \times \mathbb{1} \cong W$. 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 $c : W \to W$, we have that $(\mathrm{id}_W, c) : W \to W \times W$ and the diagonal functor $(\mathrm{id}_W, \mathrm{id}_W) : W \to W \times W$ are such that $\pi_W \circ (\mathrm{id}_W, c) = \mathrm{id}_W$ and $\pi_W \circ (\mathrm{id}_W, \mathrm{id}_W) = \mathrm{id}_W$. Hence, assuming (ii), we have that (id_W, c) and $(\mathrm{id}_W, \mathrm{id}_W)$ are inverse equivalences of π_W. Thus we have a natural isomorphism $(\mathrm{id}_W, c) \cong (\mathrm{id}_W, \mathrm{id}_W)$ which implies that

$$c \cong \pi_2 \circ (\mathrm{id}_W, c) \cong \pi_2 \circ (\mathrm{id}_W, \mathrm{id}_W) \cong \mathrm{id}_W.$$

This proves that (ii) $\Rightarrow$ (iii).

Assuming (iii), if $f, g : X \to W$ are functors, we have the natural isomorphisms

$$f = \mathrm{id}_W \circ f \cong c \circ f = c \circ g \cong \mathrm{id}_W \circ g = g.$$

This shows that (iii) $\Rightarrow$ (iv).

*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.

Finally, assuming (iv), we have that, given any functor $c : \mathbb{1} \to W$, the composition $W \to \mathbb{1} \xrightarrow{c} W$ is naturally isomorphic to the identity. Hence $W \to \mathbb{1}$ is an equivalence. ∎

Appendix B. Fibrations and indexed categories

In this section, we recall a basic aspect of the equivalence between indexed categories and fibrations. We use this result to get a better perspective over some of the properties of the Grothendieck constructions in our work.

Definition B.1 (Strictly indexed functor)**.** Let $\mathcal{L}' : \mathcal{D}^{\mathrm{op}} \to \mathbf{Cat}$ and $\mathcal{L} : \mathcal{C}^{\mathrm{op}} \to \mathbf{Cat}$ be two strictly indexed categories. A *strictly indexed functor* between $\mathcal{L}'$ and $\mathcal{L}$ consists of a pair $(\overline{H}, h)$ in which $\overline{H} : \mathcal{D} \to \mathcal{C}$ is a functor and

$$h : \mathcal{L}' \longrightarrow \left(\mathcal{L} \circ \overline{H}^{\mathrm{op}}\right) \tag{B.1}$$

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

$$\left(\overline{HE}, (h_{\overline{E}^{\mathrm{op}}}) \cdot e : \mathcal{L}'' \longrightarrow \left(\mathcal{L} \circ \left(\overline{HE}\right)^{\mathrm{op}}\right)\right). \tag{B.2}$$

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

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

$$\begin{array}{rccl} \displaystyle\int : & \mathfrak{Ind} & \to & \mathrm{Sp}\mathfrak{Fib} \\ & \mathcal{L} : \mathcal{C}^{\mathrm{op}} \to \mathbf{Cat} & \mapsto & (\mathsf{P}_{\mathcal{L}} : \Sigma_{\mathcal{C}}\mathcal{L} \to \mathcal{C}) \\ & (\overline{E}, e) & \mapsto & (E, \overline{E}) \end{array}$$

between the category of strictly indexed categories (with strictly indexed functors) and the category of (Grothendieck) split fibrations (with morphisms of split fibrations respecting the cleavage (called, in this case, splitting)).* We explicitly state the relevant part of this result below.

*Although not necessary to your work, we refer to [15] and [20, Theorem 1.3.6] for further details.

Proposition B.2. *Given two strictly indexed categories, $\mathcal{L}' : \mathcal{D}^{\mathrm{op}} \to \mathbf{Cat}$ and $\mathcal{L} : \mathcal{C}^{\mathrm{op}} \to \mathbf{Cat}$, there is a bijection between strictly indexed functors*

$$\left(\overline{H} : \mathcal{D} \to \mathcal{C}, h : \mathcal{L}' \longrightarrow \left(\mathcal{L} \circ \overline{H}^{\mathrm{op}}\right)\right) : \mathcal{L}' \to \mathcal{L}$$

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

– *The diagram*

$$\begin{tikzcd} \Sigma_{\mathcal{D}}\mathcal{L}' \arrow[r, "H"] \arrow[d, "\mathsf{P}_{\mathcal{L}'}"'] & \Sigma_{\mathcal{C}}\mathcal{L} \arrow[d, "\mathsf{P}_{\mathcal{L}}"] \\ \mathcal{D} \arrow[r, "\overline{H}"'] & \mathcal{C} \end{tikzcd} \tag{B.3}$$

commutes.

– *For any morphism $(f : X \to Y, \mathrm{id} : \mathcal{L}'(f)(y) \to \mathcal{L}'(f)(y))$ between $(X, \mathcal{L}'(f)(y))$ and (Y, y) in $\Sigma_{\mathcal{D}}\mathcal{L}'$,*

$$H(f, \mathrm{id}) = (\overline{H}(f), \mathrm{id}) : H(X, \mathcal{L}'(f)(y)) \to H(Y, y). \tag{B.4}$$

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 $(\overline{H}, h) : \mathcal{L}' \to \mathcal{L}$, we define

$$H(f : X \to Y, f' : x \to \mathcal{L}'(f)y) := (\overline{H}(f), h_X(f')). \tag{B.5}$$

Reciprocally, given a pair $(H, \overline{H})$ satisfying (B.3) and (B.4), we define

$$h_X(f' : w \to x) := H\left((\mathrm{id}_X, f') : (X, w) \to (X, x)\right) \tag{B.6}$$

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

Definition B.3 (Split fibration functor)**.** A pair $(H, \overline{H}) : \mathsf{P}_{\mathcal{L}'} \to \mathsf{P}_{\mathcal{L}}$ satisfying (B.3) and (B.4) is herein called a *split fibration functor*. Whenever it is clear from the context, we omit the split fibrations $\mathsf{P}_{\mathcal{L}'}$, $\mathsf{P}_{\mathcal{L}}$, and the functor $\overline{H}$.

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

$$\int \mathcal{L} \;=\; (\mathsf{P}_{\mathcal{L}} : \Sigma_{\mathcal{C}}\mathcal{L} \to \mathcal{C})$$

$$\int \left(\overline{H}, h\right) \;=\; \left(H, \overline{H}\right)$$

in which $H\left(f : X \to Y, f' : x \to \mathcal{L}(f)(y)\right) = \left(\overline{H}(f), h_X(f')\right)$.

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

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

It should be noted that

$$\left(\int \mathcal{L}' \underline{\times} \mathcal{L}\right) \cong \left(\int \mathcal{L}'\right) \times \left(\int \mathcal{L}\right) = \left(\mathsf{P}_{\mathcal{L}'} \times \mathsf{P}_{\mathcal{L}} : (\Sigma_{\mathcal{D}}\mathcal{L}') \times (\Sigma_{\mathcal{C}}\mathcal{L}) \to \mathcal{D} \times \mathcal{C}\right), \tag{B.7}$$

which means that the product in $\mathfrak{SpFib}$ coincides with the usual product of functors $\mathsf{P}_{\mathcal{L}} \times \mathsf{P}_{\mathcal{L}'}$. Moreover, given indexed functors $(\overline{H}, h) : \mathcal{H} \to \mathcal{H}'$ and $(\overline{E}, e) : \mathcal{L} \to \mathcal{L}'$, we have that

$$(\overline{H}, h) \underline{\times} (\overline{E}, e) = \left(\overline{H} \times \overline{E}, h \times e\right)$$

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

$$\int \left((\overline{H}, h) \underline{\times} (\overline{E}, e)\right) = \left(\int (\overline{H}, h)\right) \times \left(\int (\overline{E}, e)\right). \tag{B.8}$$

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

$$\int^{\mathrm{op}} \mathcal{L} = \left(\mathsf{P}_{\mathcal{L}^{\mathrm{op}}} : \Sigma_{\mathcal{C}}\mathcal{L}^{\mathrm{op}} \to \mathcal{C}\right), \qquad \int^{\mathrm{op}} \left(\overline{H}, h\right) = \left(H, \overline{H}\right)$$

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

Appendix C. Coproducts in the total category

In this section, we recall general results about coproducts and initial objects in the total categories of fibrations.

Proposition C.1 (Initial object in $\Sigma_{\mathcal{C}}\mathcal{L}$)**.** *Let $\mathcal{L} : \mathcal{C}^{\mathrm{op}} \to \mathbf{Cat}$ be a strictly indexed category. We assume that*

– *$\mathcal{C}$ has initial object $\mathbb{0}$;*
– *$\mathcal{L}(\mathbb{0})$ 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 $(Y, y) \in \Sigma_{\mathcal{C}}\mathcal{L}$,

$$
\begin{aligned}
&\Sigma_{\mathcal{C}}\mathcal{L}\left((\mathbb{0},\mathbb{0}),(Y,y)\right) \\
&= \coprod_{n\in\mathcal{C}(\mathbb{0},Y)} \mathcal{L}(\mathbb{0})(\mathbb{0},\mathcal{L}(n)(y)) && \left\{ \text{ Definition } \right\} \\
&\cong \coprod_{n\in\mathcal{C}(\mathbb{0},Y)} \mathbb{1} && \left\{ \mathbb{0} \text{ initial in } \mathcal{L}(\mathbb{0}) \right\} \\
&\cong \mathbb{1}. && \left\{ \mathbb{0} \text{ initial in } \mathcal{C} \right\}
\end{aligned}
$$

■

Theorem C.1 (Coproducts in $\Sigma_{\mathcal{C}}\mathcal{L}$)**.** *Let $\mathcal{L} : \mathcal{C}^{\mathrm{op}} \to \mathbf{Cat}$ be a strictly indexed category. We assume that*

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

$$\left(W_t \xrightarrow{\iota_{W_t}} \coprod_{i\in I} W_i \right)_{t\in I} \tag{C.1}$$

of the objects in $((W_i, w_i))_{i\in I}$;
– *there is an adjunction $\mathcal{L}(\iota_{W_i})! \dashv \mathcal{L}(\iota_{W_i})$ for each $i \in I$;*
– *$\mathcal{L}\left(\coprod_{i\in I} W_i\right)$ 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 $(Y, y) \in \Sigma_{\mathcal{C}}\mathcal{L}$,

$$\prod_{i\in I} \Sigma_{\mathcal{C}}\mathcal{L}\left((W_i, w_i),(Y,y)\right)$$

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

■

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

Corollary C.2 (Initial object in $\Sigma_{\mathcal{C}}\mathcal{L}^{\mathrm{op}}$)**.** *Let $\mathcal{L}:\mathcal{C}^{\mathrm{op}}\to\mathbf{Cat}$ be a strictly indexed category. We assume that*

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

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

Corollary C.3 (Coproducts in $\Sigma_{\mathcal{C}}\mathcal{L}^{\mathrm{op}}$)**.** *Let $\mathcal{L}:\mathcal{C}^{\mathrm{op}}\to\mathbf{Cat}$ be a strictly indexed category. We assume that*

- *$((W_i,w_i))_{i\in I}$ is family of objects of $\Sigma_{\mathcal{C}}\mathcal{L}$;*

– *the category* $\mathcal{C}$ *has the coproduct*

$$\left(W_t \xrightarrow{\iota_{W_t}} \coprod_{i\in I} W_i \right)_{t\in I} \tag{C.2}$$

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

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

– $\mathcal{L}\left(\coprod_{i\in I} W_i\right)$ *has the product*

$$\prod_{i\in I} \mathcal{L}(\iota_{W_i})^*(w_i)$$

of the objects $(\mathcal{L}(\iota_{W_i})^*(w_i))_{i\in I}$.

In this case,

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

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

Appendix D. Parameterized initial algebras

This section aims to recall the basic aspects of the constructions related to parameterized initial algebras.

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

$$\begin{tikzcd} E(W) \arrow[r, "E(g)"] \arrow[d, "\zeta"'] & E(Y) \arrow[d, "\xi"] \\ W \arrow[r, "g"'] & Y \end{tikzcd} \tag{D.1}$$

commutes. Dually, we define the category E-CoAlg of E-coalgebras by

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

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

Recall that, provided that they exist, the initial object $(\mu E, \mathfrak{in}_E)$ of E-Alg and the terminal object $(\nu E, \mathfrak{out}_E)$ of E-CoAlg are such that $\mathfrak{in}_E$ and $\mathfrak{out}_E$ are invertible. In this setting, we denote by

$$\mathrm{fold}_E(Y,\xi) : \mu E \to Y, \quad \mathrm{unfold}_E(X,\varrho) : X \to \nu E \tag{D.3}$$

the unique morphisms in $\mathcal{D}$ such that

$$\begin{tikzcd}
E(\mu E) \arrow[r, "E(\mathrm{fold}_E(Y,\xi))"] \arrow[d, "\mathfrak{in}_E"'] & E(Y) \arrow[d, "\xi"] \\
\mu E \arrow[r, "\mathrm{fold}_E(Y,\xi)"'] & Y
\end{tikzcd}
\qquad
\begin{tikzcd}
X \arrow[r, "\mathrm{unfold}_E(X,\varrho)"] \arrow[d, "\varrho"'] & \nu E \arrow[d, "\mathfrak{out}_E"] \\
E(X) \arrow[r, "E(\mathrm{unfold}_E(X,\varrho))"'] & E(\nu E)
\end{tikzcd}
\tag{D.4}$$

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

We recall below how to explicitly construct the parameterized initial algebras and terminal coalgebras.

Proposition D.1 (μ-operator and ν-operator)**.** *Let $H : \mathcal{D}' \times \mathcal{D} \to \mathcal{D}$ 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*

$$\begin{array}{rcl}
\mu H : \mathcal{D}' & \to & \mathcal{D} \\
X & \mapsto & \mu H^X \\
(f : X \to Y) & \mapsto & \mathrm{fold}_{H^X}\left(\mathfrak{in}_{H^Y} \circ H(f, \mu H^Y)\right).
\end{array}$$

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

$$\begin{array}{rcl}
\nu H : \mathcal{D}' & \to & \mathcal{D} \\
X & \mapsto & \nu H^X \\
(f : X \to Y) & \mapsto & \mathrm{unfold}_{H^Y}\left(H(f, \nu H^X) \circ \mathfrak{out}_{H^X}\right).
\end{array}$$

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

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

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

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

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

and, hence, the diagram

$$\begin{tikzcd}
H\left(X, \mu H^X\right) \arrow[r, "H(X,\mu H(g)\circ\mu H(f))"] \arrow[d, "\mathfrak{in}_{H^X}"'] & H\left(X, \mu H^Z\right) \arrow[d, "\mathfrak{in}_{H^Z}\circ H\left(g\circ f,\mu H^Z\right)"] \\
\mu H^X \arrow[r, "\mu H(g)\circ\mu H(f)"'] & \mu H^Z
\end{tikzcd}$$

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

$$\begin{aligned}
&\mu H(g) \circ \mu H(f) \\
&= \mathrm{fold}_{H^X}\left(\mathfrak{in}_{H^Z} \circ H\left(g \circ f, \mu H^Z\right)\right) \\
&= \mu H(g \circ f). && \left\{\ \text{definition}\ \right\}
\end{aligned}$$

■

It should be noted that $\mathcal{D}'$ above is any category. However, Proposition D.1 usually is considered in the setting in which $\mathcal{D}' = \mathcal{D}^{n-1}$ for $n > 1$ as described below.

Proposition D.2 (Parameterized initial algebras and terminal coalgebras)**.** *Let $H : \mathcal{D}^n \to \mathcal{D}$ 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*

$$\begin{array}{rcl} \mu H : \mathcal{D}^{n-1} & \to & \mathcal{D} \\ X & \mapsto & \mu H^X \\ (f : X \to Y) & \mapsto & \mathrm{fold}_{H^X}\left(\mathfrak{in}_{H^Y} \circ H(f, \mu H^Y)\right). \end{array}$$

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

$$\begin{array}{rcl} \nu H : \mathcal{D}^{n-1} & \to & \mathcal{D} \\ X & \mapsto & \nu H^X \\ (f : X \to Y) & \mapsto & \mathrm{unfold}_{H^Y}\left(H(f, \nu H^X) \circ \mathfrak{out}_{H^X}\right). \end{array}$$

Appendix E. Preservation, reflection and creation of initial algebras

We establish the definitions of creation, preservation and reflection of initial algebras and terminal cooalgebras.

Lemma E.1. *Let $F : \mathcal{D} \to \mathcal{C}$ be a functor. Given endofunctors $E : \mathcal{C} \to \mathcal{C}$, $E' : \mathcal{D} \to \mathcal{D}$ and a natural transformation $\gamma : E \circ F \longrightarrow F \circ E'$, we have an induced functor defined by*

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

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

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

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 $\check{F}_\gamma$ follows, then, from that of F. ∎

Dually, we have:

Lemma E.2. *Let $E : \mathcal{C} \to \mathcal{C}$, $G : \mathcal{C} \to \mathcal{D}$, and $E' : \mathcal{D} \to \mathcal{D}$ be functors. Each natural transformation $\beta : G \circ E \longrightarrow E' \circ G$ induces a functor*

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

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 E.3 (Preservation, reflection and creation of initial algebras)**.** We say that a functor $F : \mathcal{D} \to \mathcal{C}$ *(strictly)*[*] *preserves the initial algebra/reflects the initial algebra/creates the initial algebra* of the endofunctor $E : \mathcal{C} \to \mathcal{C}$ if, whenever $E' : \mathcal{D} \to \mathcal{D}$ is such that $\gamma : E \circ F \cong F \circ E'$ (or, in the strict case, $F \circ E' = E \circ F$), the functor

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

induced by γ (induced by the identity) is such that it (strictly) preserves the initial object/reflects the initial object/creates the initial object.

Remark E.4. *A functor $F : \mathcal{D} \to \mathcal{C}$ (strictly) preserves the initial algebra of the endofunctor $E : \mathcal{D} \to \mathcal{D}$ if, and only if, for any natural isomorphism $\gamma : E \circ F \cong F \circ E'$ (or, in the strict case, for any identity $E \circ F = F \circ E'$) in which E' is an endofunctor, we have that E-*Alg *has an initial object whenever E'-*Alg *does, and*

$$\mathrm{fold}_E \left(F(\mu E'), F\left(\mathfrak{in}_{E'}\right) \circ \gamma_{\mu E'} \right) : \mu E \to F\left(\mu E'\right) \tag{E.1}$$

is an isomorphism (the identity).

Definition E.5 (Preservation, reflection and creation of initial algebras)**.** We say that a functor $F : \mathcal{D} \to \mathcal{C}$ *(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 $\mathcal{D}$.

[*]Whenever we talk about strict preservation, we are assuming that we have chosen initial objects/terminal objects.

Remark E.6. *In other words, let $F : \mathcal{D} \to \mathcal{C}$ be a functor.*

1. *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 $(\mu E', \mathfrak{in}_{E'})$ is the initial E'-algebra, the E-algebra $\left(F(\mu E'), F(\mathfrak{in}_{E'}) \circ \gamma_{\mu E'}\right)$ is an initial object of* E-Alg *(the chosen initial object of* E-Alg*, in the strict case).*
2. *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.*
3. *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'-Alg *has an initial algebra if* E-Alg *does.*

Definition E.7 (Preservation, reflection and creation of terminal coalgebras)**.** We say that a functor $G : \mathcal{C} \to \mathcal{D}$ *(strictly) preserves the initial algebra/reflects the initial algebra/creates the initial algebra* of an endofunctor $E : \mathcal{C} \to \mathcal{C}$ if, for any natural isomorphism $\beta : G \circ E \cong E' \circ G$, the functor

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

induced by β (induced by the identity) is such that it (strictly) preserves the terminal object/reflects the terminal object/creates the terminal object.

Definition E.8 (Preservation, reflection and creation of terminal coalgebras)**.** We say that a functor $G : \mathcal{C} \to \mathcal{D}$ *(strictly) preserves initial algebras/reflects initial algebras/creates initial algebras* if G (strictly) preserves initial algebras/reflects initial algebras/creates initial algebras of any endofunctor $E : \mathcal{C} \to \mathcal{C}$.

E.1. Indexed categories. This subsection aims to reach a suitable notion of what it means for an indexed category to respect initial algebras and terminal coalgebras. These notions play a fundamental role in our approach to study parameterized initial algebras and final coalgebras in the total category of a split fibration (Corollary F.1).

Definition E.9 (Respecting initial algebras)**.** A strictly indexed category $\mathcal{L} : \mathcal{C}^{\mathrm{op}} \to \mathbf{Cat}$ *respects initial algebras* if $\mathcal{L}(f)$ strictly preserves initial algebras for any morphism f of $\mathcal{C}$.*

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

Appendix F. Parameterized initial algebras for split fibrations

In this section, we establish and prove general results about parameterized initial algebras on the total category of a split fibration. We start by introducing a basic result on endofunctors.

Theorem F.1 (Initial algebras of strictly indexed endofunctors)**.** *Let $(\overline{E}, e)$ be a strictly indexed endofunctor on $\mathcal{L} : \mathcal{C}^{\mathrm{op}} \to \mathbf{Cat}$ and $E : \Sigma_{\mathcal{C}}\mathcal{L} \to \Sigma_{\mathcal{C}}\mathcal{L}$ the corresponding split fibration endofunctor. Assume that*

- *the initial $\overline{E}$-algebra $\left(\mu\overline{E},\, \mathfrak{in}_{\overline{E}}\right)$ exists;*
- *the initial $\left(\mathcal{L}(\mathfrak{in}_{\overline{E}})^{-1} e_{\mu\overline{E}}\right)$-algebra*

$$\left(\mu\left(\mathcal{L}(\mathfrak{in}_{\overline{E}})^{-1} e_{\mu\overline{E}}\right),\, \mathfrak{in}_{\left(\mathcal{L}(\mathfrak{in}_{\overline{E}})^{-1} e_{\mu\overline{E}}\right)}\right)$$

exists.

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

$$\mu E = \left(\mu\overline{E},\, \mu\underline{e}\right), \qquad \mathfrak{in}_E = \left(\mathfrak{in}_{\overline{E}},\, \mathcal{L}(\mathfrak{in}_{\overline{E}})\left(\mathfrak{in}_{\underline{e}}\right)\right). \tag{F.1}$$

Moreover, for each E-algebra

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

we have that

$$\mathrm{fold}_E\left(\xi,\xi'\right) = \left(\mathrm{fold}_{\overline{E}}\xi,\ \mathrm{fold}_{\underline{e}}\left(\mathcal{L}\left(\overline{E}(\mathrm{fold}_{\overline{E}}\,\xi)\cdot \mathfrak{in}_{\overline{E}}^{-1}\right)(\xi')\right)\right). \tag{F.2}$$

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

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

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

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

*We could have allowed non-strict preservation but, in our context, it is more practical to keep things as strict as possible when it comes to strict indexed categories.

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

$$\begin{tikzcd}
\underline{e}(\mu\underline{e}) \arrow[r, "\underline{e}\left(\mathrm{fold}_{\underline{e}}\,\mathcal{L}\left(\overline{E}(\mathrm{fold}_{\overline{E}}\xi)\cdot\mathfrak{in}_{\overline{E}}^{-1}\right)(\xi')\right)"] \arrow[ddd, "\mathfrak{in}_{\underline{e}}"'] & \underline{e}\circ\mathcal{L}\left(\mathrm{fold}_{\overline{E}}\xi\right)(y) \arrow[d, equal] \\
 & \mathcal{L}\left(\overline{E}(\mathrm{fold}_{\overline{E}}\xi)\cdot\mathfrak{in}_{\overline{E}}^{-1}\right)\circ e_Y(y) \arrow[d, "\mathcal{L}\left(\overline{E}(\mathrm{fold}_{\overline{E}}\xi)\cdot\mathfrak{in}_{\overline{E}}^{-1}\right)(\xi')"] \\
 & \mathcal{L}\left(\xi\cdot\overline{E}(\mathrm{fold}_{\overline{E}}\xi)\cdot\mathfrak{in}_{\overline{E}}^{-1}\right)(y) \arrow[d, equal] \\
\mu\underline{e} \arrow[r, "\left(\mathrm{fold}_{\underline{e}}\,\mathcal{L}\left(\overline{E}(\mathrm{fold}_{\overline{E}}\xi)\cdot\mathfrak{in}_{\overline{E}}^{-1}\right)(\xi')\right)"'] & \mathcal{L}\left(\mathrm{fold}_{\overline{E}}\xi\right)(y)
\end{tikzcd}$$

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

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

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

$$\begin{tikzcd}
e_{\mu\overline{E}}(\mu\underline{e}) \arrow[r, "e_{\mu\overline{E}}\left(\mathrm{fold}_{\underline{e}}\,\mathcal{L}\left(\overline{E}(\mathrm{fold}_{\overline{E}}\xi)\cdot\mathfrak{in}_{\overline{E}}^{-1}\right)(\xi')\right)"] \arrow[ddd, "\mathcal{L}(\mathfrak{in}_{\overline{E}})(\mathfrak{in}_{\underline{e}})"'] & e_{\mu\overline{E}}\circ\mathcal{L}\left(\mathrm{fold}_{\overline{E}}\xi\right)(y) \arrow[d, equal] \\
 & \mathcal{L}\left(\overline{E}(\mathrm{fold}_{\overline{E}}\xi)\right)\circ e_Y(y) \arrow[d, "\mathcal{L}\left(\overline{E}(\mathrm{fold}_{\overline{E}}\xi)\right)(\xi')"] \\
 & \mathcal{L}\left(\overline{E}(\mathrm{fold}_{\overline{E}}\xi)\right)\circ\mathcal{L}\left(\xi\right)(y) \arrow[d, equal] \\
\mathcal{L}(\mathfrak{in}_{\overline{E}})(\mu\underline{e}) \arrow[r, "\mathcal{L}(\mathfrak{in}_{\overline{E}})\left(\mathrm{fold}_{\underline{e}}\,\mathcal{L}\left(\overline{E}(\mathrm{fold}_{\overline{E}}\xi)\cdot\mathfrak{in}_{\overline{E}}^{-1}\right)(\xi')\right)"'] & \mathcal{L}\left(\mathfrak{in}_{\overline{E}}\right)\circ\mathcal{L}\left(\mathrm{fold}_{\overline{E}}\xi\right)(y)
\end{tikzcd}$$

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

$$\mathfrak{u}=\left(\mathrm{fold}_{\overline{E}}\xi,\ \left(\mathrm{fold}_{\underline{e}}\,\mathcal{L}\left(\overline{E}(\mathrm{fold}_{\overline{E}}\xi)\cdot\mathfrak{in}_{\overline{E}}^{-1}\right)(\xi')\right)\right) \tag{F.3}$$

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

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

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

Let $\mathcal{L} : \mathcal{C}^{\mathrm{op}} \to \mathbf{Cat}$, $\mathcal{L}' : \mathcal{D}^{\mathrm{op}} \to \mathbf{Cat}$ be strictly indexed categories as above. We denote by $\mathcal{L}' \underline{\times} \mathcal{L} : (\mathcal{D} \times \mathcal{C})^{\mathrm{op}} \to \mathbf{Cat}$ the product of the indexed categories (see Appendix B). An object of $\Sigma_{\mathcal{D}\times\mathcal{C}}\,(\mathcal{L}' \underline{\times} \mathcal{L}) \cong (\Sigma_{\mathcal{D}}\mathcal{L}') \times (\Sigma_{\mathcal{C}}\mathcal{L})$ 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

$$(f, g) : (X_0, W_0) \to (X_1, W_1)$$

is a morphism in $\mathcal{D} \times \mathcal{C}$, and

$$(f', g') : (x_0, w_0) \to (\mathcal{L}'(f)(x_1), \mathcal{L}(g)(w_1))$$

is a morphism in $\mathcal{L}'(X_0) \times \mathcal{L}(W_0)$.

Given a strictly indexed functor $(\overline{H}, h) : \mathcal{L}' \underline{\times} \mathcal{L} \to \mathcal{L}$ 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 $h^{(X,x)} : \mathcal{L} \longrightarrow \left(\mathcal{L} \circ \overline{H}^X\right)$ is pointwise defined by

$$\begin{array}{rcl} h_Y^{(X,x)} : & \mathcal{L}(Y) & \to \mathcal{L} \circ \overline{H}^X(Y) \\ & f' : y \to z & \mapsto h_{(X,Y)}(x, f') \end{array}$$

in which we denote by $(X, Y) \in \mathcal{D} \times \mathcal{C}$. To be consistent with the notation previously introduced (in Proposition D.1), we also denote by $h^x_{(X,Y)}$ the morphism $h_Y^{(X,x)}$ above.

As a consequence of Theorem F.1, we have that, under suitable conditions, parameterized initial algebras of split fibration functors are split fibration functors. Namely, we have:

Theorem F.2 (Parameterized initial algebras are split fibration functors). *Let $(\overline{H}, h)$ be a strictly indexed functor from $\mathcal{L}' \underline{\times} \mathcal{L} : (\mathcal{D} \times \mathcal{C})^{\mathrm{op}} \to \mathbf{Cat}$ to $\mathcal{L} : \mathcal{C}^{\mathrm{op}} \to \mathbf{Cat}$, and*

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

the corresponding split fibration functor. Assume that

– *for each object X of $\mathcal{D}$, the initial $\overline{H}^X$-algebra $\left(\mu\overline{H}^X, \mathfrak{in}_{\overline{H}^X}\right)$ exists;*
– *for each object (X,x) in $\Sigma_{\mathcal{D}}\mathcal{L}'$, denoting by $\underline{h}_X$ the functor*

$$\mathcal{L}(\mathfrak{in}_{\overline{H}^X})^{-1} h_{\left(X,\mu\overline{H}^X\right)} : \mathcal{L}'(X) \times \mathcal{L}(\mu\overline{H}^X) \to \mathcal{L}(\mu\overline{H}^X) \tag{F.4}$$

is such that the initial $\underline{h}_X^x$-algebra $\left(\mu\underline{h}_X^x, \mathfrak{in}_{\underline{h}_X^x}\right)$ exists;
– *for each morphism $g : X \to Y$ in $\mathcal{D}$ and $y \in \mathcal{L}'(Y)$, the equation*

$$\mathcal{L}\left(\mu\overline{H}(g)\right)(\mathfrak{in}_{\underline{h}_Y^y}) = \mathfrak{in}_{\underline{h}_X^{\mathcal{L}'(g)(y)}} \tag{F.5}$$

holds.

In this setting, the parameterized initial algebra $\mu H : \Sigma_{\mathcal{D}}\mathcal{L}' \to \Sigma_{\mathcal{C}}\mathcal{L}$ 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 F.1. Hence we have that

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

exists by Proposition D.1. More precisely, given a morphism $(f,f') : (X,x) \to (Y,y)$ in $\Sigma_{\mathcal{D}}\mathcal{L}'$, we compute $\mu H(f,f')$ below.

$$\begin{aligned}
&\mu H(f,f') \\
&= \mathrm{fold}_{H^{(X,x)}}\left(\mathfrak{in}_{H^{(Y,y)}} \circ H\left((f,f'), \mu H^{(Y,y)}\right)\right) && \{\text{ Proposition D.1 }\} \\
&= \mathrm{fold}_{H^{(X,x)}}\left(\left(\mathfrak{in}_{\overline{H}^Y}, \mathcal{L}(\mathfrak{in}_{\overline{H}^Y})(\mathfrak{in}_{\underline{h}_Y^y})\right) \circ H\left((f,f'), \mu H^{(Y,y)}\right)\right) && \{\text{ Eq. (F.1) }\} \\
&= \mathrm{fold}_{H^{(X,x)}}\left(\left(\mathfrak{in}_{\overline{H}^Y}, \mathcal{L}(\mathfrak{in}_{\overline{H}^Y})(\mathfrak{in}_{\underline{h}_Y^y})\right) \circ \left(\overline{H}(f,\mu\overline{H}^Y), h_{(X,\mu\overline{H}^Y)}(f',\mu\underline{h}_Y^y)\right)\right) && \{\text{ indexed functor }\} \\
&= \mathrm{fold}_{H^{(X,x)}}\left(\mathfrak{in}_{\overline{H}^Y} \circ \overline{H}(f,\mu\overline{H}^Y), \mathcal{L}\left(\mathfrak{in}_{\overline{H}^Y} \circ \overline{H}(f,\mu\overline{H}^Y)\right)(\mathfrak{in}_{\underline{h}_Y^y})\right. \\
&\qquad\qquad \left.\circ \left(h_{(X,\mu\overline{H}^Y)}(f',\mu\underline{h}_Y^y)\right)\right) && \{\text{ composing }\}
\end{aligned}$$

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

$$\begin{aligned}
&\mathrm{fold}_{H^{(X,x)}}\left(\mathfrak{in}_{\overline{H}^Y} \circ \overline{H}(f,\mu\overline{H}^Y), \xi'\right) \\
&= \left(\mathrm{fold}_{\overline{H}^X}\left(\mathfrak{in}_{\overline{H}^Y} \circ \overline{H}(f,\mu\overline{H}^Y)\right), \left(\mathrm{fold}_{\underline{h}_X^x} \mathcal{L}\left(\overline{H}^X(\mathrm{fold}_{\overline{H}^X}\xi) \cdot \mathfrak{in}_{\overline{H}^X}^{-1}\right)(\xi')\right)\right) && \{\text{ Eq. (F.2) }\} \\
&= \left(\mu\overline{H}(f), \left(\mathrm{fold}_{\underline{h}_X^x} \mathcal{L}\left(\overline{H}^X(\mathrm{fold}_{\overline{H}^X}\xi) \cdot \mathfrak{in}_{\overline{H}^X}^{-1}\right)(\xi')\right)\right). && \{\text{ Proposition D.1 }\}
\end{aligned}$$

The above shows that

$$\mu H(f,f') = \left(\mu\overline{H}(f), \left(\mathrm{fold}_{\underline{h}_X^x} \mathcal{L}\left(\overline{H}^X(\mathrm{fold}_{\overline{H}^X}\xi) \cdot \mathfrak{in}_{\overline{H}^X}^{-1}\right)(\xi')\right)\right). \tag{F.6}$$

Now, we can proceed to prove that μH is actually a split fibration functor. Firstly, by Equation (F.6), we have that

$$\begin{tikzcd}
\Sigma_{\mathcal{D}}\mathcal{L}' \arrow[rr, "\mu H"] \arrow[d, "\mathsf{P}_{\mathcal{L}'}"'] & & \Sigma_{\mathcal{C}}\mathcal{L} \arrow[d, "\mathsf{P}_{\mathcal{L}}"] \\
\mathcal{D} \arrow[rr, "\mu\overline{H}"'] & & \mathcal{C}
\end{tikzcd} \tag{F.7}$$

commutes.

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

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

we have that

$$\begin{aligned}
&\left(\mathrm{fold}_{\underline{h}_X^{\mathcal{L}'(g)(y)}}\, \mathcal{L}\left(\overline{H}^X(\mathrm{fold}_{\overline{H}^X}\xi)\cdot \mathfrak{in}_{\overline{H}^X}^{-1}\right)(\xi')\right) \\
&= \left(\mathrm{fold}_{\underline{h}_X^{\mathcal{L}'(g)(y)}}\, \mathcal{L}\left(\xi \cdot \overline{H}^X(\mathrm{fold}_{\overline{H}^X}\xi)\cdot \mathfrak{in}_{\overline{H}^X}^{-1}\right)\left(\mathfrak{in}_{\underline{h}_Y^y}\right)\right) && \left\{\ h_{(X,\mu\overline{H}^Y)}(\mathrm{id}, \mu\underline{h}_Y^y) = \mathrm{id}\ \right\} \\
&= \left(\mathrm{fold}_{\underline{h}_X^{\mathcal{L}'(g)(y)}}\, \mathcal{L}\left((\mathrm{fold}_{\overline{H}^X}\xi)\cdot \mathfrak{in}_{\overline{H}^X}\cdot \mathfrak{in}_{\overline{H}^X}^{-1}\right)\left(\mathfrak{in}_{\underline{h}_Y^y}\right)\right) && \left\{\ \mathrm{fold}_{\overline{H}^X}\xi\ \right\} \\
&= \left(\mathrm{fold}_{\underline{h}_X^{\mathcal{L}'(g)(y)}}\, \mathcal{L}\left(\mu\overline{H}(g)\right)\left(\mathfrak{in}_{\underline{h}_Y^y}\right)\right) && \left\{\ \text{Proposition D.1}\ \right\} \\
&= \mathrm{id}_{\mu\underline{h}_X^{\mathcal{L}'(g)(y)}} && \left\{\ \text{Eq. (F.5)}\ \right\}
\end{aligned}$$

By Equation (F.6), the above proves that

$$\mu H\left(g, \mathrm{id}\right) = \left(\mu\overline{H}(g), \mathrm{id}\right)$$

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 F.3 (Parameterized initial algebras are strictly indexed functors)**.** *Let $(\overline{H}, h)$ be a strictly indexed functor from $\mathcal{L}' \underline{\times} \mathcal{L} : (\mathcal{D} \times \mathcal{C})^{\mathrm{op}} \to \mathbf{Cat}$ to $\mathcal{L} : \mathcal{C}^{\mathrm{op}} \to \mathbf{Cat}$, and $H : (\Sigma_{\mathcal{D}}\mathcal{L}') \times (\Sigma_{\mathcal{C}}\mathcal{L}) \to \Sigma_{\mathcal{C}}\mathcal{L}$ the corresponding split fibration functor. Assume that:*

– *the parameterized initial algebra $\mu\overline{H} : \mathcal{D} \to \mathcal{C}$ exists;*

– *for any $X \in \mathcal{D}$, the parameterized initial algebra $\mu\underline{h}_X$ exists;*
– *for each morphism $g : X \to Y$ in $\mathcal{D}$ and $y \in Y$, the equation*

$$\mathcal{L}\left(\mu\overline{H}(g)\right)\left(\mathfrak{in}_{\underline{h}_Y^y}\right) = \mathfrak{in}_{\underline{h}_X^{\mathcal{L}'(g)(y)}} \tag{F.8}$$

holds.

In this setting, the parameterized initial algebra

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

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

$$\mu\left(\underline{h}_{(X)}\right) = \mu\underline{h}_X = \mu\left(\mathcal{L}(\mathfrak{in}_{\overline{H}^X})^{-1} h_{\left(X, \mu\overline{H}^X\right)}\right) : \mathcal{L}'(X) \to \mathcal{L}(\mu\overline{H}^X). \tag{F.9}$$

Proof: By Theorem F.2 (Eq. (F.6)) and Proposition B.2 (Eq. (B.5)), we have that

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

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

$$\begin{aligned}
&\mathfrak{h}_X(f') \\
&= \mu H(\mathrm{id}_X, f') \\
&= \left(\mathrm{id}_{\mu\overline{H}^X}, \mathrm{fold}_{\underline{h}_X^x}\left(\mathfrak{in}_{\underline{h}_X^w} \circ \mathcal{L}\left(\mathfrak{in}^{-1}_{\overline{H}^X}\right)\left(h_{\left(X, \mu\overline{H}^X\right)}(f', \mu\underline{h}_X^w)\right)\right)\right) && \{ \text{ Eq. (F.6) } \} \\
&= \left(\mathrm{id}_{\mu\overline{H}^X}, \mathrm{fold}_{\underline{h}_X^x}\left(\mathfrak{in}_{\underline{h}_X^w} \circ \underline{h}_X\left(f', \mu\underline{h}_X^w\right)\right)\right) \\
&= \left(\mathrm{id}_{\mu\overline{H}^X}, \mu\underline{h}_X(f')\right) && \{ \text{ Proposition D.1 } \}
\end{aligned}$$

■

Finally, for strictly indexed categories respecting initial algebras (see Definition E.9), we get a cleaner version of Theorem F.3 below.

Corollary F.1 (Parameterized initial algebras and strictly indexed categories respecting initial algebras)**.** *Let $(\overline{H}, h)$ be a strictly indexed functor from $\mathcal{L}' \underline{\times} \mathcal{L} : (\mathcal{D} \times \mathcal{C})^{\mathrm{op}} \to \mathbf{Cat}$ to $\mathcal{L} : \mathcal{C}^{\mathrm{op}} \to \mathbf{Cat}$, and $H : (\Sigma_{\mathcal{D}}\mathcal{L}') \times (\Sigma_{\mathcal{C}}\mathcal{L}) \to \Sigma_{\mathcal{C}}\mathcal{L}$ the corresponding split fibration functor. Assume that:*

– *$\mathcal{L}$ respects initial algebras;*
– *the parameterized initial algebra $\mu\overline{H} : \mathcal{D} \to \mathcal{C}$ exists;*
– *for any $X \in \mathcal{D}$, the parameterized initial algebra $\mu\underline{h}_X$ exists.*

In this setting, the parameterized initial algebra

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

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

$$\mu\left(\underline{h}_{(X)}\right) = \mu\underline{h}_X = \mu\left(\mathcal{L}(\mathfrak{in}_{\overline{H}^X})^{-1} h_{\left(X, \mu\overline{H}^X\right)}\right) : \mathcal{L}'(X) \to \mathcal{L}(\mu\overline{H}^X). \tag{F.10}$$

Proof: By Theorem F.3, it is enough to show that Equation (F.8) holds whenever $\mathcal{L}$ respects initial algebras.

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

$$\begin{tikzcd}
\mathcal{L}\left(\mu\overline{H}^Y\right) \arrow[r, "\mathcal{L}\left(\mu\overline{H}(g)\right)"] \arrow[d, "\left(y, \mathrm{id}_{\mathcal{L}(\mu\overline{H}^Y)}\right)"'] & \mathcal{L}(\mu\overline{H}^X) \arrow[d, "\left(\mathcal{L}'(y), \mathrm{id}_{\mathcal{L}(\mu\overline{H}^X)}\right)"] \\
\mathcal{L}'(Y) \times \mathcal{L}\left(\mu\overline{H}^Y\right) \arrow[r, "\mathcal{L}'(g) \times \mathcal{L}\left(\mu\overline{H}(g)\right)"] \arrow[d, "h_{\left(Y, \mu\overline{H}^Y\right)}"'] & \mathcal{L}'(X) \times \mathcal{L}\left(\mu\overline{H}^X\right) \arrow[d, "h_{\left(X, \mu\overline{H}^X\right)}"] \\
\mathcal{L}\left(\overline{H}\left(Y, \mu\overline{H}^Y\right)\right) \arrow[r, "\mathcal{L}\left(\overline{H}\left(g, \mu\overline{H}(g)\right)\right)"] \arrow[d, "\mathcal{L}\left(\mathfrak{in}_{\overline{H}^Y}\right)^{-1}"'] & \mathcal{L}\left(\overline{H}\left(X, \mu\overline{H}^X\right)\right) \arrow[d, "\mathcal{L}\left(\mathfrak{in}_{\overline{H}^X}\right)^{-1}"] \\
\mathcal{L}\left(\mu\overline{H}^Y\right) \arrow[r, "\mathcal{L}\left(\mu\overline{H}(g)\right)"'] & \mathcal{L}\left(\mu\overline{H}^X\right)
\end{tikzcd}$$

commute. Thus, we get that

$$\begin{aligned}
&\mathcal{L}\left(\mu\overline{H}(g)\right) \circ \underline{h}_Y^y \\
&= \mathcal{L}\left(\mu\overline{H}(g)\right) \circ \underline{h}_Y \circ \left(y, \mathrm{id}_{\mathcal{L}(\mu\overline{H}^Y)}\right) \\
&= \mathcal{L}\left(\mu\overline{H}(g)\right) \circ \mathcal{L}\left(\mathfrak{in}_{\overline{H}^Y}\right)^{-1} \circ h_{\left(Y, \mu\overline{H}^Y\right)} \circ \left(y, \mathrm{id}_{\mathcal{L}(\mu\overline{H}^Y)}\right)
\end{aligned}$$

$$= \mathcal{L}\left(\mathfrak{in}_{\overline{H}^X}\right)^{-1} \circ h_{\left(X,\mu\overline{H}^X\right)} \circ \left(\mathcal{L}'(y), \mathrm{id}_{\mathcal{L}(\mu\overline{H}^X)}\right) \circ \mathcal{L}\left(\mu\overline{H}(g)\right)$$
$$= \underline{h}_X^{\mathcal{L}'(y)} \circ \mathcal{L}\left(\mu\overline{H}(g)\right).$$

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

$$\mathcal{L}\left(\mu\overline{H}(g)\right)\left(\mathfrak{in}_{\underline{h}_Y^y}\right) = \mathfrak{in}_{\underline{h}_X^{\mathcal{L}'(g)(y)}}$$

holds. That is to say (F.8) holds for any $g : X \to Y$ in $\mathcal{D}$ and any $y \in \mathcal{L}'(Y)$. This completes the proof by Theorem F.3. ∎

Appendix G. Parameterized terminal coalgebras for split fibrations

In this section, we establish and prove general results about terminal coalgebras of endofunctors on the total category of split fibrations.

Definition E.5 about strictly indexed categories respecting terminal coalgebras plays a central role in our basic result below.

Theorem G.1 (Terminal coalgebras of strictly indexed endofunctors)**.** *Let $(\overline{E}, e)$ be a strictly indexed endofunctor on $\mathcal{L} : \mathcal{C}^{\mathrm{op}} \to \mathbf{Cat}$ and $E : \Sigma_{\mathcal{C}}\mathcal{L} \to \Sigma_{\mathcal{C}}\mathcal{L}$ the corresponding split fibration endofunctor. Assume that*

- *$\mathcal{L}$ respects terminal coalgebras;*
- *the terminal $\overline{E}$-coalgebra $\left(\nu\overline{E}, \mathfrak{out}_{\overline{E}}\right)$ exists;*
- *the terminal $(\mathcal{L}(\mathfrak{out}_{\overline{E}})e_{\nu\overline{E}})$-coalgebra $\left(\nu\left(\mathcal{L}(\mathfrak{out}_{\overline{E}})e_{\nu\overline{E}}\right), \mathfrak{out}_{\mathcal{L}(\mathfrak{out}_{\overline{E}})e_{\nu\overline{E}}}\right)$ exists.*

Denoting by $\overline{e}$ the endofunctor $\mathcal{L}\left(\mathfrak{out}_{\overline{E}}\right)e_{\nu\overline{E}}$ on $\mathcal{L}(\nu\overline{E})$, the terminal E-coalgebra exists and is given by

$$\nu E = \left(\nu\overline{E}, \nu\overline{e}\right), \qquad \mathfrak{out}_E = \left(\mathfrak{out}_{\overline{E}}, \mathfrak{out}_{\overline{e}}\right). \tag{G.1}$$

Moreover, for each E-coalgebra

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

we have that

$$\mathrm{unfold}_E(\xi,\xi') = \left(\mathrm{unfold}_{\overline{E}}\xi,\ \mathrm{unfold}_{\mathcal{L}(\xi)e_Y}\xi'\right). \tag{G.2}$$

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

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

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

$$\begin{tikzcd}
\mathcal{L}\left(\nu\overline{E}\right) \arrow[r, "\mathcal{L}(\mathrm{unfold}_{\overline{E}}\xi)"] \arrow[d, "e_{\nu\overline{E}}"'] \arrow[dd, bend right=60, "\overline{e}"'] & \mathcal{L}\left(Y\right) \arrow[d, "e_Y"] \\
\mathcal{L}\left(\overline{E}\left(\nu\overline{E}\right)\right) \arrow[r, "\mathcal{L}\left(\overline{E}(\mathrm{unfold}_{\overline{E}}\xi)\right)"] \arrow[d, "\mathcal{L}(\mathfrak{out}_{\overline{E}})"'] & \mathcal{L}\left(\overline{E}\left(Y\right)\right) \arrow[d, "\mathcal{L}(\xi)"] \\
\mathcal{L}\left(\nu\overline{E}\right) \arrow[r, "\mathcal{L}(\mathrm{unfold}_{\overline{E}}\xi)"'] & \mathcal{L}\left(Y\right)
\end{tikzcd}$$

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

$$\left(\mathcal{L}\left(\mathrm{unfold}_{\overline{E}}\xi\right)\left(\nu\overline{e}\right),\ \mathcal{L}\left(\mathrm{unfold}_{\overline{E}}\xi\right)\left(\mathfrak{out}_{\overline{e}}\right)\right)$$

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

$$\mathrm{unfold}_{\mathcal{L}(\xi)e_Y}\xi' : y \to \mathcal{L}\left(\mathrm{unfold}_{\overline{E}}\xi\right)\left(\nu\overline{e}\right)$$

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

$$\begin{tikzcd}
y \arrow[r, "\mathrm{unfold}_{\mathcal{L}(\xi)e_Y}\xi'"] \arrow[d, "\xi'"'] & \mathcal{L}\left(\mathrm{unfold}_{\overline{E}}\xi\right)\left(\nu\overline{e}\right) \arrow[d, "\mathcal{L}(\mathrm{unfold}_{\overline{E}}\xi)(\mathfrak{out}_{\overline{e}})"] \\
\mathcal{L}\left(\xi\right)e_Y\left(y\right) \arrow[r, "\mathcal{L}(\xi)e_Y\left(\mathrm{unfold}_{\mathcal{L}(\xi)e_Y}\xi'\right)"'] & \mathcal{L}\left(\xi\right)e_Y\mathcal{L}\left(\mathrm{unfold}_{\overline{E}}\xi\right)\left(\nu\overline{e}\right)
\end{tikzcd}$$

which shows that

$$\left(\mathrm{unfold}_{\overline{E}}\xi,\ \mathrm{unfold}_{\mathcal{L}(\xi)e_Y}\xi'\right) : (Y, y) \to E(Y, y) = \left(\overline{E}(Y), e_Y(y)\right)$$

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

$$\begin{tikzcd}
(Y, y) \arrow[r, "\left(\mathrm{unfold}_{\overline{E}}\xi,\ \mathrm{unfold}_{\mathcal{L}(\xi)e_Y}\xi'\right)"] \arrow[d, "(\xi,\xi')"'] & \left(\nu\overline{E},\ \nu\overline{e}\right) \arrow[d, "\left(\mathfrak{out}_{\overline{E}},\ \mathfrak{out}_{\overline{e}}\right)"] \\
E\left(Y, y\right) = \left(\overline{E}(Y), e_Y(y)\right) \arrow[r, "\left(\overline{E}\left(\mathrm{unfold}_{\overline{E}}\xi\right), e_Y\left(\mathrm{unfold}_{\mathcal{L}(\xi)e_Y}\xi'\right)\right)", "E\left(\mathrm{unfold}_{\overline{E}}\xi,\ \mathrm{unfold}_{\mathcal{L}(\xi)e_Y}\xi'\right)"'] & E\left(\nu\overline{E},\ \nu\overline{e}\right) = \left(\overline{E}\left(\nu\overline{E}\right), \overline{e}\left(\nu\overline{e}\right)\right)
\end{tikzcd}$$

commutes. This completes the proof that $\nu E = \left(\nu\overline{E},\, \nu\overline{e}\right)$ is the terminal E-coalgebra. ∎

Theorem G.2 (Parameterized terminal coalgebras are strictly indexed functors)**.** *Let $(\overline{H}, h)$ be a strictly indexed functor from $\mathcal{L}' \underline{\times} \mathcal{L} : (\mathcal{D} \times \mathcal{C})^{\mathrm{op}} \to \mathbf{Cat}$ to $\mathcal{L} : \mathcal{C}^{\mathrm{op}} \to \mathbf{Cat}$, and $H : (\Sigma_{\mathcal{D}}\mathcal{L}') \times (\Sigma_{\mathcal{C}}\mathcal{L}) \to \Sigma_{\mathcal{C}}\mathcal{L}$ the corresponding split fibration functor. Assume that*

- *$\mathcal{L}$ respects terminal coalgebras;*
- *for each object X of $\mathcal{C}$, the terminal $\overline{H}^X$-coalgebra $\left(\nu\overline{H}^X,\, \mathsf{out}_{\overline{H}^X}\right)$ exists;*
- *for each object (X, x) in $\Sigma_{\mathcal{D}}\mathcal{L}'$, denoting by $\overline{h}_X$ the functor*
$$\mathcal{L}(\mathsf{out}_{\overline{H}^X}) h_{\left(X, \nu\overline{H}^X\right)} : \mathcal{L}'(X) \times \mathcal{L}(\nu\overline{H}^X) \to \mathcal{L}(\nu\overline{H}^X) \tag{G.3}$$
is such that the terminal $\overline{h}^x_X$-coalgebra $\left(\nu\overline{h}^x_X,\, \mathsf{out}_{\overline{h}^x_X}\right)$ exists.

In this setting, the parameterized terminal coalgebra
$$\nu H : \Sigma_{\mathcal{D}}\mathcal{L}' \to \Sigma_{\mathcal{C}}\mathcal{L}$$
is a split fibration functor coming from the strictly indexed functor $\left(\nu\overline{H}, \nu\left(\overline{h}_{(-)}\right)\right)$ in which, for each $X \in \mathcal{D}$,
$$\nu\left(\overline{h}_{(X)}\right) = \nu\overline{h}_X = \nu\left(\mathcal{L}(\mathsf{out}_{\overline{H}^X}) h_{\left(X, \nu\overline{H}^X\right)}\right) : \mathcal{L}'(X) \to \mathcal{L}(\nu\overline{H}^X). \tag{G.4}$$

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 G.1. Hence, by Proposition D.1, we have that
$$\nu H : \Sigma_{\mathcal{D}}\mathcal{L}' \to \Sigma_{\mathcal{C}}\mathcal{L}$$
exists. More precisely, given a morphism $(f, f') : (X, x) \to (Y, y)$ in $\Sigma_{\mathcal{D}}\mathcal{L}'$, we compute $\nu H(f, f')$ below.

$$
\begin{aligned}
&\nu H(f, f') \\
&= \mathrm{unfold}_{H^{(Y,y)}}\left(H\left((f, f'), \nu H^{(X,x)}\right) \circ \mathsf{out}_{H^{(X,x)}}\right) && \{\text{ Proposition D.1 }\} \\
&= \mathrm{unfold}_{H^{(Y,y)}}\left(H\left((f, f'), \nu H^{(X,x)}\right) \circ \left(\mathsf{out}_{\overline{H}^X},\, \mathsf{out}_{\overline{h}^x_X}\right)\right) && \{\text{ Eq. (G.1) }\} \\
&= \mathrm{unfold}_{H^{(Y,y)}}\left(\left(\overline{H}(f, \nu\overline{H}^X), h_{\left(X, \nu\overline{H}^X\right)}(f', \nu\overline{h}^x_X)\right) \circ \left(\mathsf{out}_{\overline{H}^X},\, \mathsf{out}_{\overline{h}^x_X}\right)\right) && \{\text{ hypothesis }\} \\
&= \mathrm{unfold}_{H^{(Y,y)}}\left(\overline{H}(f, \nu\overline{H}^X) \circ \mathsf{out}_{\overline{H}^X},\, \mathcal{L}\left(\mathsf{out}_{\overline{H}^X}\right)\left(h_{\left(X, \nu\overline{H}^X\right)}(f', \nu\overline{h}^x_X)\right) \circ \mathsf{out}_{\overline{h}^x_X}\right) && \{\text{ composing }\} \\
&= \mathrm{unfold}_{H^{(Y,y)}}\left(\overline{H}(f, \nu\overline{H}^X) \circ \mathsf{out}_{\overline{H}^X},\, \overline{h}_X\left(f', \nu\overline{h}^x_X\right) \circ \mathsf{out}_{\overline{h}^x_X}\right) && \{\text{ definition of } \overline{h}_X \}
\end{aligned}
$$

$$= \left(\mathrm{unfold}_{\overline{H}^Y}\left(\overline{H}(f, \nu\overline{H}^X) \circ \mathtt{out}_{\overline{H}^X}\right), \mathrm{unfold}_{\overline{h}^y_Y}\left(\overline{h}_X\left(f', \nu\overline{h}^x_X\right) \circ \mathtt{out}_{\overline{h}^x_X}\right)\right) \qquad \{ \text{ Eq. (G.2) } \}$$

$$= \left(\nu\overline{H}(f), \nu\overline{h}_Y(f')\right) \qquad \{ \text{ Proposition D.1 } \}$$

Since $\nu H(f, f') = \left(\nu\overline{H}(f), \nu\overline{h}_Y(f')\right)$, clearly, then, the pair $\left(\nu H, \nu\overline{H}\right)$ satisfies Eq. (B.3) and Eq. (B.4) of Proposition B.3. Moreover, νH comes from the strictly indexed functor $\left(\nu\overline{H}, \nu\left(\overline{h}_{(-)}\right)\right)$. ■

Appendix H. CHAD transformation without sharing between primal and (co)tangents

In this section, we list the CHAD program transformations $\overrightarrow{\mathcal{D}}_{\overline{\Gamma}}(t)_1$, $\overrightarrow{\mathcal{D}}_{\overline{\Gamma}}(t)_2$, $\overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(t)_1$ and $\overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(t)_2$ of a program t 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.

H.1. Forward-mode AD.

$$\overrightarrow{\mathcal{D}}_{\overline{\Gamma}}(\mathrm{op}(t_1, \ldots, t_k))_1 \stackrel{\mathrm{def}}{=} \mathbf{let}\, x_1 = \overrightarrow{\mathcal{D}}_{\overline{\Gamma}}(t_1)_1\, \mathbf{in} \cdots \mathbf{let}\, x_k = \overrightarrow{\mathcal{D}}_{\overline{\Gamma}}(t_k)_1\, \mathbf{in}\, \mathrm{op}(x_1, \ldots, x_k)$$

$$\overrightarrow{\mathcal{D}}_{\overline{\Gamma}}(x)_1 \stackrel{\mathrm{def}}{=} x$$

$$\overrightarrow{\mathcal{D}}_{\overline{\Gamma}}(\mathbf{let}\, x = t\, \mathbf{in}\, s)_1 \stackrel{\mathrm{def}}{=} \mathbf{let}\, x = \overrightarrow{\mathcal{D}}_{\overline{\Gamma}}(t)_1\, \mathbf{in}\, \overrightarrow{\mathcal{D}}_{\overline{\Gamma},x}(s)_1$$

$$\overrightarrow{\mathcal{D}}_{\overline{\Gamma}}(\langle\rangle)_1 \stackrel{\mathrm{def}}{=} \langle\rangle$$

$$\overrightarrow{\mathcal{D}}_{\overline{\Gamma}}(\langle t, s\rangle)_1 \stackrel{\mathrm{def}}{=} \langle\overrightarrow{\mathcal{D}}_{\overline{\Gamma}}(t)_1, \overrightarrow{\mathcal{D}}_{\overline{\Gamma}}(s)_1\rangle$$

$$\overrightarrow{\mathcal{D}}_{\overline{\Gamma}}(\mathbf{fst}\,(t))_1 \stackrel{\mathrm{def}}{=} \mathbf{fst}\,(\overrightarrow{\mathcal{D}}_{\overline{\Gamma}}(t)_1)$$

$$\overrightarrow{\mathcal{D}}_{\overline{\Gamma}}(\mathbf{snd}\,(t))_1 \stackrel{\mathrm{def}}{=} \mathbf{snd}\,(\overrightarrow{\mathcal{D}}_{\overline{\Gamma}}(t)_1)$$

$$\overrightarrow{\mathcal{D}}_{\overline{\Gamma}}(\lambda x.t)_1 \stackrel{\mathrm{def}}{=} \lambda x.\langle\overrightarrow{\mathcal{D}}_{\overline{\Gamma},x}(t)_1, \underline{\lambda}\mathsf{v}.\mathbf{let}\,\mathsf{v} = \langle\underline{0}, \mathsf{v}\rangle\, \mathbf{in}\, \overrightarrow{\mathcal{D}}_{\overline{\Gamma},x}(t)_2\rangle$$

$$\overrightarrow{\mathcal{D}}_{\overline{\Gamma}}(t\, s)_1 \stackrel{\mathrm{def}}{=} \mathbf{fst}\,(\overrightarrow{\mathcal{D}}_{\overline{\Gamma}}(t)_1\, \overrightarrow{\mathcal{D}}_{\overline{\Gamma}}(s)_1)$$

$$\overrightarrow{\mathcal{D}}_{\overline{\Gamma}}(\ell t)_1 \stackrel{\mathrm{def}}{=} \ell(\overrightarrow{\mathcal{D}}_{\overline{\Gamma}}(t)_1)$$

$$\overrightarrow{\mathcal{D}}_{\overline{\Gamma}}(\mathbf{case}\, t\, \mathbf{of}\, \{\ell_1 x_1 \to s_1 \mid \cdots \mid \ell_n x_n \to s_n\})_1 \stackrel{\mathrm{def}}{=}$$
$$\mathbf{case}\, \overrightarrow{\mathcal{D}}_{\overline{\Gamma}}(t)_1\, \mathbf{of}\, \{\ell_1 x_1 \to \overrightarrow{\mathcal{D}}_{\overline{\Gamma},x_1}(s_1)_1 \mid \cdots \mid \ell_n x_n \to \overrightarrow{\mathcal{D}}_{\overline{\Gamma},x_n}(s_n)_1\}$$

$$\overrightarrow{\mathcal{D}}_{\overline{\Gamma}}(\mathbf{roll}\, t)_1 \stackrel{\mathrm{def}}{=} \mathbf{roll}\, \overrightarrow{\mathcal{D}}_{\overline{\Gamma}}(t)_1$$

$$\overrightarrow{\mathcal{D}}_{\overline{\Gamma}}(\mathbf{fold}\, t\, \mathbf{with}\, x \to s)_1 \stackrel{\mathrm{def}}{=} \mathbf{fold}\, \overrightarrow{\mathcal{D}}_{\overline{\Gamma}}(t)_1\, \mathbf{with}\, x \to \overrightarrow{\mathcal{D}}_x(s)_1$$

$$\overrightarrow{\mathcal{D}}_{\overline{\Gamma}}(\mathbf{gen\,from}\,t\,\mathbf{with}\,x \to s)_1 \stackrel{\mathrm{def}}{=} \mathbf{gen\,from}\,\overrightarrow{\mathcal{D}}_{\overline{\Gamma}}(t)_1\,\mathbf{with}\,x \to \overrightarrow{\mathcal{D}}_{x}(s)_1$$

$$\overrightarrow{\mathcal{D}}_{\overline{\Gamma}}(\mathbf{unroll}\,t)_1 \stackrel{\mathrm{def}}{=} \mathbf{unroll}\,\overrightarrow{\mathcal{D}}_{\overline{\Gamma}}(t)_1$$

$$\begin{aligned}
\overrightarrow{\mathcal{D}}_{\overline{\Gamma}}(\mathrm{op}(t_1,\ldots,t_k))_2 &\stackrel{\mathrm{def}}{=} \mathbf{let}\,x_1 = \overrightarrow{\mathcal{D}}_{\overline{\Gamma}}(t_1)_1\,\mathbf{in}\cdots\mathbf{let}\,x_k = \overrightarrow{\mathcal{D}}_{\overline{\Gamma}}(t_k)_1\,\mathbf{in}\\
&\qquad D\mathrm{op}(x_1,\ldots,x_k;\langle \overrightarrow{\mathcal{D}}_{\overline{\Gamma}}(t_1)_2 \bullet \mathsf{v},\ldots,\overrightarrow{\mathcal{D}}_{\overline{\Gamma}}(t_k)\bullet_2 \mathsf{v}\rangle)
\end{aligned}$$

$$\overrightarrow{\mathcal{D}}_{\overline{\Gamma}}(x)_2 \stackrel{\mathrm{def}}{=} \mathbf{proj}_{\mathbf{idx}(x;\overline{\Gamma})}\,(\mathsf{v})$$

$$\overrightarrow{\mathcal{D}}_{\overline{\Gamma}}(\mathbf{let}\,x = t\,\mathbf{in}\,s)_2 \stackrel{\mathrm{def}}{=} \mathbf{let}\,x = \overrightarrow{\mathcal{D}}_{\overline{\Gamma}}(t)_1\,\mathbf{in}\,\mathbf{let}\,\mathsf{v} = \langle \mathsf{v}, \overrightarrow{\mathcal{D}}_{\overline{\Gamma}}(t)_2\rangle\,\mathbf{in}\,\overrightarrow{\mathcal{D}}_{\overline{\Gamma},x}(s)_2$$

$$\overrightarrow{\mathcal{D}}_{\overline{\Gamma}}(\langle\rangle)_2 \stackrel{\mathrm{def}}{=} \langle\rangle$$

$$\overrightarrow{\mathcal{D}}_{\overline{\Gamma}}(\langle t,s\rangle)_2 \stackrel{\mathrm{def}}{=} \langle \overrightarrow{\mathcal{D}}_{\overline{\Gamma}}(t)_2, \overrightarrow{\mathcal{D}}_{\overline{\Gamma}}(s)_2\rangle$$

$$\overrightarrow{\mathcal{D}}_{\overline{\Gamma}}(\mathbf{fst}\,(t))_2 \stackrel{\mathrm{def}}{=} \mathbf{fst}\,(\overrightarrow{\mathcal{D}}_{\overline{\Gamma}}(t)_2)$$

$$\overrightarrow{\mathcal{D}}_{\overline{\Gamma}}(\mathbf{snd}\,(t))_2 \stackrel{\mathrm{def}}{=} \mathbf{snd}\,(\overrightarrow{\mathcal{D}}_{\overline{\Gamma}}(t)_2)$$

$$\overrightarrow{\mathcal{D}}_{\overline{\Gamma}}(\lambda x.t)_2 \stackrel{\mathrm{def}}{=} \lambda x.\mathbf{let}\,\mathsf{v} = \langle \mathsf{v}, \underline{0}\rangle\,\mathbf{in}\,\overrightarrow{\mathcal{D}}_{\overline{\Gamma},x}(t)_2$$

$$\overrightarrow{\mathcal{D}}_{\overline{\Gamma}}(t\,s)_2 \stackrel{\mathrm{def}}{=} \mathbf{let}\,y = \overrightarrow{\mathcal{D}}_{\overline{\Gamma}}(s)_1\,\mathbf{in}\,\overrightarrow{\mathcal{D}}_{\overline{\Gamma}}(t)_2\,y + (\mathbf{snd}\,(\overrightarrow{\mathcal{D}}_{\overline{\Gamma}}(t)_1\,y)) \bullet \overrightarrow{\mathcal{D}}_{\overline{\Gamma}}(s)_2$$

$$\overrightarrow{\mathcal{D}}_{\overline{\Gamma}}(\ell t)_2 \stackrel{\mathrm{def}}{=} \overrightarrow{\mathcal{D}}_{\overline{\Gamma}}(t)_2$$

$$\begin{aligned}
&\overrightarrow{\mathcal{D}}_{\overline{\Gamma}}(\mathbf{case}\,t\,\mathbf{of}\,\{\ell_1 x_1 \to s_1 \mid \cdots \mid \ell_n x_n \to s_n\})_2 \stackrel{\mathrm{def}}{=}\\
&\qquad \mathbf{bunch}\,(\overrightarrow{\mathcal{D}}_{\overline{\Gamma}}(t)_1, \langle \mathsf{v}, \overrightarrow{\mathcal{D}}_{\overline{\Gamma}}(t)_2\rangle)\,\mathbf{of}\,\{(\ell_1 x_1,\mathsf{v}) \to \overrightarrow{\mathcal{D}}_{\overline{\Gamma},x_1}(s_1)_2 \mid \cdots \mid (\ell_n x_n,\mathsf{v}) \to \overrightarrow{\mathcal{D}}_{\overline{\Gamma},x_n}(s_n)_2\}
\end{aligned}$$

$$\overrightarrow{\mathcal{D}}_{\overline{\Gamma}}(\mathbf{roll}\,t)_2 \stackrel{\mathrm{def}}{=} \mathbf{roll}\,\overrightarrow{\mathcal{D}}_{\overline{\Gamma}}(t)_2$$

$$\begin{aligned}
&\overrightarrow{\mathcal{D}}_{\overline{\Gamma}}(\mathbf{fold}\,t\,\mathbf{with}\,x \to s)_2 \stackrel{\mathrm{def}}{=} \mathbf{fold}\,\overrightarrow{\mathcal{D}}_{\overline{\Gamma}}(t)_2\,\mathbf{with}\,\mathsf{v} \to\\
&\qquad \mathbf{let}\,x = \mathbf{fold}\,\overrightarrow{\mathcal{D}}_{\overline{\Gamma}}(t)_1\,\mathbf{with}\,x \to \overrightarrow{\mathcal{D}}(\tau)_1[{}^{x\vdash \overrightarrow{\mathcal{D}}_x(s)_1}/_{\alpha}]\,\mathbf{in}\,\overrightarrow{\mathcal{D}}_x(s)_2
\end{aligned}$$

$$\overrightarrow{\mathcal{D}}_{\overline{\Gamma}}(\mathbf{gen\,from}\,t\,\mathbf{with}\,x \to s)_2 \stackrel{\mathrm{def}}{=} \mathbf{gen\,from}\,\overrightarrow{\mathcal{D}}_{\overline{\Gamma}}(t)_2\,\mathbf{with}\,\mathsf{v} \to \mathbf{let}\,x = \overrightarrow{\mathcal{D}}_{\overline{\Gamma}}(t)_1\,\mathbf{in}\,\overrightarrow{\mathcal{D}}_x(s)_2$$

$$\overrightarrow{\mathcal{D}}_{\overline{\Gamma}}(\mathbf{unroll}\,t)_2 \stackrel{\mathrm{def}}{=} \mathbf{unroll}\,\overrightarrow{\mathcal{D}}_{\overline{\Gamma}}(t)_2$$

H.2. Reverse-mode AD.

$$\overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(\mathrm{op}(t_1,\ldots,t_k))_1 \stackrel{\mathrm{def}}{=} \mathbf{let}\,x_1 = \overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(t_1)\,\mathbf{in}\cdots\mathbf{let}\,x_k = \overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(t_k)\,\mathbf{in}\,\mathrm{op}(x_1)$$

$$\overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(x)_1 \stackrel{\mathrm{def}}{=} x$$

$$\overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(\mathbf{let}\,x = t\,\mathbf{in}\,s)_1 \stackrel{\mathrm{def}}{=} \mathbf{let}\,x = \overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(t)_1\,\mathbf{in}\,\overleftarrow{\mathcal{D}}_{\overline{\Gamma},x}(s)_1$$

$$\overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(\langle\rangle)_1 \stackrel{\mathrm{def}}{=} \langle\rangle$$

$$\overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(\langle t,s\rangle)_1 \stackrel{\mathrm{def}}{=} \langle \overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(t)_1, \overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(s)_1\rangle$$

$$\overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(\mathbf{fst}\,(t))_1 \stackrel{\mathrm{def}}{=} \mathbf{fst}\,(\overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(t)_1)$$

$$\overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(\mathbf{snd}\,(t))_1 \stackrel{\mathrm{def}}{=} \mathbf{snd}\,(\overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(t)_1)$$
$$\overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(\lambda x.t)_1 \stackrel{\mathrm{def}}{=} \lambda x.\langle \overleftarrow{\mathcal{D}}_{\overline{\Gamma},x}(t)_1, \underline{\lambda}\mathsf{v}.\mathbf{snd}\,(\overleftarrow{\mathcal{D}}_{\overline{\Gamma},x}(t)_2)\rangle$$
$$\overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(t\,s)_1 \stackrel{\mathrm{def}}{=} \mathbf{fst}\,(\overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(t)_1\,\overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(s)_1)$$
$$\overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(\ell t)_1 \stackrel{\mathrm{def}}{=} \ell(\overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(t)_1)$$
$$\overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(\mathbf{case}\,t\,\mathbf{of}\,\{\ell_1 x_1 \to s_1 \mid \cdots \mid \ell_n x_n \to s_n\})_1 \stackrel{\mathrm{def}}{=}$$
$$\quad \mathbf{case}\,\overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(t)_1\,\mathbf{of}\,\{\ell_1 x_1 \to \overleftarrow{\mathcal{D}}_{\overline{\Gamma},x_1}(s_1)_1 \mid \cdots \mid \ell_n x_n \to \overleftarrow{\mathcal{D}}_{\overline{\Gamma},x_n}(s_n)_1\}$$
$$\overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(\mathbf{roll}\,t)_1 \stackrel{\mathrm{def}}{=} \mathbf{roll}\,\overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(t)_1$$
$$\overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(\mathbf{fold}\,t\,\mathbf{with}\,x \to s)_1 \stackrel{\mathrm{def}}{=} \mathbf{fold}\,\overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(t)_1\,\mathbf{with}\,x \to \overleftarrow{\mathcal{D}}_{x}(s)_1$$
$$\overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(\mathbf{gen\,from}\,t\,\mathbf{with}\,x \to s)_1 \stackrel{\mathrm{def}}{=} \mathbf{gen\,from}\,\overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(t)_1\,\mathbf{with}\,x \to \overleftarrow{\mathcal{D}}_{x}(s)_1$$
$$\overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(\mathbf{unroll}\,t)_1 \stackrel{\mathrm{def}}{=} \mathbf{unroll}\,\overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(t)_1$$

$$\overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(\mathrm{op}(t_1,\ldots,t_k))_2 \stackrel{\mathrm{def}}{=} \mathbf{let}\,x_1 = \overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(t_1)\,\mathbf{in}\,\cdots\mathbf{let}\,x_k = \overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(t_k)\,\mathbf{in}\,\mathbf{let}\,\mathsf{v} = D\mathrm{op}^t(x_1,\ldots,x_k;\mathsf{v})\,\mathbf{in}$$
$$(\mathbf{let}\,\mathsf{v} = \mathbf{proj}_1\,\mathsf{v}\,\mathbf{in}\,\overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(t_1)_2) + \cdots + (\mathbf{let}\,\mathsf{v} = \mathbf{proj}_1\,\mathsf{v}\,\mathbf{in}\,\overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(t_k)_2)$$
$$\overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(x)_2 \stackrel{\mathrm{def}}{=} \mathbf{coproj}_{\mathbf{idx}(x;\overline{\Gamma})}\,(\mathsf{v})$$
$$\overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(\mathbf{let}\,x = t\,\mathbf{in}\,s)_2 \stackrel{\mathrm{def}}{=} \mathbf{let}\,x = \overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(t)_1\,\mathbf{in}\,\mathbf{let}\,\mathsf{v} = \overleftarrow{\mathcal{D}}_{\overline{\Gamma},x}(s)_2\,\mathbf{in}\,\mathbf{fst}\,(\mathsf{v}) + \mathbf{let}\,\mathsf{v} = \mathbf{snd}\,(\mathsf{v})\,\mathbf{in}\,\overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(t)_2$$
$$\overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(\langle\rangle)_2 \stackrel{\mathrm{def}}{=} \underline{0}$$
$$\overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(\langle t,s\rangle)_2 \stackrel{\mathrm{def}}{=} (\mathbf{let}\,\mathsf{v} = \mathbf{fst}\,(\mathsf{v})\,\mathbf{in}\,\overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(t)_2) + (\mathbf{let}\,\mathsf{v} = \mathbf{snd}\,(\mathsf{v})\,\mathbf{in}\,\overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(s)_2)$$
$$\overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(\mathbf{fst}\,(t))_2 \stackrel{\mathrm{def}}{=} \mathbf{let}\,\mathsf{v} = \langle \mathsf{v}, \underline{0}\rangle\,\mathbf{in}\,\overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(t)_2$$
$$\overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(\mathbf{snd}\,(t))_2 \stackrel{\mathrm{def}}{=} \mathbf{let}\,\mathsf{v} = \langle \underline{0}, \mathsf{v}\rangle\,\mathbf{in}\,\overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(t)_2$$
$$\overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(\lambda x.t)_2 \stackrel{\mathrm{def}}{=} \mathbf{case}\,\mathsf{v}\,\mathbf{of}\,!x \otimes \mathsf{v} \to \mathbf{fst}\,(\overleftarrow{\mathcal{D}}_{\overline{\Gamma},x}(t)_2)$$
$$\overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(t\,s)_2 \stackrel{\mathrm{def}}{=} \mathbf{let}\,x = \overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(s)_1\,\mathbf{in}\,(\mathbf{let}\,\mathsf{v} = !x \otimes \mathsf{v}\,\mathbf{in}\,\overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(t)_2) +$$
$$(\mathbf{let}\,\mathsf{v} = (\mathbf{snd}\,(\overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(t)_1\,x)) \bullet \mathsf{v}\,\mathbf{in}\,\overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(s)_2)$$
$$\overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(\ell t)_2 \stackrel{\mathrm{def}}{=} \overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(t)_2$$
$$\overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(\mathbf{case}\,t\,\mathbf{of}\,\{\ell_1 x_1 \to s_1 \mid \cdots \mid \ell_n x_n \to s_n\})_2 \stackrel{\mathrm{def}}{=}$$
$$\quad \mathbf{let}\,\mathsf{v} = \mathbf{bunch}\,(\overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(t)_1, \mathsf{v})\,\mathbf{of}\,\{(\ell_1 x_1, \mathsf{v}) \to \overleftarrow{\mathcal{D}}_{\overline{\Gamma},x_1}(s_1)_2 \mid \cdots \mid (\ell_n x_n, \mathsf{v}) \to \overleftarrow{\mathcal{D}}_{\overline{\Gamma},x_n}(s_n)_2\}\,\mathbf{in}$$
$$\quad \mathbf{fst}\,\mathsf{v} + \mathbf{let}\,\mathsf{v} = \mathbf{snd}\,\mathsf{v}\,\mathbf{in}\,\overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(t)_2$$
$$\overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(\mathbf{roll}\,t)_2 \stackrel{\mathrm{def}}{=} \mathbf{let}\,\mathsf{v} = \mathbf{unroll}\,\mathsf{v}\,\mathbf{in}\,\overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(t)_2$$
$$\overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(\mathbf{fold}\,t\,\mathbf{with}\,x \to s)_2 \stackrel{\mathrm{def}}{=} \mathbf{let}\,\mathsf{v} = \big(\mathbf{gen\,from}\,\mathsf{v}\,\mathbf{with}\,\mathsf{v} \to$$

$$\mathbf{let}\, x = \ \mathbf{fold}\, \overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(t)_1\, \mathbf{with}\, x \to \overleftarrow{\mathcal{D}}(\tau)_1[{}^{x\vdash \overleftarrow{\mathcal{D}}_x(s)_1}/_{\alpha}]\, \mathbf{in}\, \overleftarrow{\mathcal{D}}_x(s)_2)\, \mathbf{in}\, \overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(t)_2$$

$$\overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(\mathbf{gen\, from}\, t\, \mathbf{with}\, x \to s)_2 \overset{\mathrm{def}}{=} \mathbf{let}\, \mathsf{v} = \ (\mathbf{fold}\, \mathsf{v}\, \mathbf{with}\, \mathsf{v} \to \mathbf{let}\, x = \ \overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(t)_1\, \mathbf{in}\, \overleftarrow{\mathcal{D}}_x(s)_2)\, \mathbf{in}\, \overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(t)_2$$

$$\overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(\mathbf{unroll}\, t)_2 \overset{\mathrm{def}}{=} \mathbf{let}\, \mathsf{v} = \ \mathbf{roll}\, \mathsf{v}\, \mathbf{in}\, \overleftarrow{\mathcal{D}}_{\overline{\Gamma}}(t)_2$$

Appendix I. 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 [40, Appendix A], reusing their notations.

For coproducts, we have the following constructs in the source language:

$$\begin{aligned} \mathbf{inl} &\in \mathbf{Syn}(\tau, \tau + \sigma) \\ \mathbf{inr} &\in \mathbf{Syn}(\sigma, \tau + \sigma) \\ [,] : \mathbf{Syn}(\tau, \rho) \times \mathbf{Syn}(\sigma, \rho) &\to \mathbf{Syn}(\tau + \sigma, \rho). \end{aligned}$$

Forward AD. We have, in the applied target language **ALSyn**, that

$$\begin{aligned}
&\overrightarrow{\mathcal{D}}(\mathbf{inl}\,)_1 \in \mathbf{ALSyn}(\overrightarrow{\mathcal{D}}(\tau)_1, \overrightarrow{\mathcal{D}}(\tau)_1 + \overrightarrow{\mathcal{D}}(\tau)_2) \\
&\overrightarrow{\mathcal{D}}(\mathbf{inl}\,)_2 \in \mathbf{ALSyn}(\overrightarrow{\mathcal{D}}(\tau)_1, \mathbf{LFun}(\overrightarrow{\mathcal{D}}(\tau)_2, \overrightarrow{\mathcal{D}}(\tau)_2 {*} \overrightarrow{\mathcal{D}}(\sigma)_2)) \\
&\overrightarrow{\mathcal{D}}(\mathbf{inr}\,)_1 \in \mathbf{ALSyn}(\overrightarrow{\mathcal{D}}(\sigma)_1, \overrightarrow{\mathcal{D}}(\tau)_1 + \overrightarrow{\mathcal{D}}(\tau)_2) \\
&\overrightarrow{\mathcal{D}}(\mathbf{inr}\,)_2 \in \mathbf{ALSyn}(\overrightarrow{\mathcal{D}}(\sigma)_1, \mathbf{LFun}(\overrightarrow{\mathcal{D}}(\sigma)_2, \overrightarrow{\mathcal{D}}(\tau)_2 {*} \overrightarrow{\mathcal{D}}(\sigma)_2)) \\
&\overrightarrow{\mathcal{D}}([t, s])_1 \in \mathbf{ALSyn}(\overrightarrow{\mathcal{D}}(\tau)_1 + \overrightarrow{\mathcal{D}}(\sigma)_1, \overrightarrow{\mathcal{D}}(\rho)_1) \\
&\overrightarrow{\mathcal{D}}([t, s])_2 \in \mathbf{ALSyn}(\overrightarrow{\mathcal{D}}(\tau)_1 + \overrightarrow{\mathcal{D}}(\sigma)_1, \mathbf{LFun}(\overrightarrow{\mathcal{D}}(\tau)_2 {*} \overrightarrow{\mathcal{D}}(\sigma)_2, \overrightarrow{\mathcal{D}}(\rho)_2)).
\end{aligned}$$

We can define

$$\begin{aligned}
\overrightarrow{\mathcal{D}}(\tau + \sigma)_1 &\overset{\mathrm{def}}{=} \overrightarrow{\mathcal{D}}(\tau)_1 + \overrightarrow{\mathcal{D}}(\sigma)_1 \\
\overrightarrow{\mathcal{D}}(\tau + \sigma)_2 &\overset{\mathrm{def}}{=} \overrightarrow{\mathcal{D}}(\tau)_1 {*} \overrightarrow{\mathcal{D}}(\sigma)_1 \\
\overrightarrow{\mathcal{D}}(\mathbf{inl}\,)_1 &\overset{\mathrm{def}}{=} \mathbf{inl} \\
\overrightarrow{\mathcal{D}}(\mathbf{inl}\,)_2 &\overset{\mathrm{def}}{=} \lambda_.\mathbf{lpair}(\mathbf{lid}, \underline{0}) \\
\overrightarrow{\mathcal{D}}(\mathbf{inr}\,)_1 &\overset{\mathrm{def}}{=} \mathbf{inr} \\
\overrightarrow{\mathcal{D}}(\mathbf{inr}\,)_2 &\overset{\mathrm{def}}{=} \lambda_.\mathbf{lpair}(\underline{0}, \mathbf{lid})
\end{aligned}$$

$$\overrightarrow{\mathcal{D}}([t,s])_1 \stackrel{\text{def}}{=} x \vdash \mathbf{case}\, x\, \mathbf{of}\, \{\mathbf{inl}\ x \to \overrightarrow{\mathcal{D}}(t)_1 | x \to \overrightarrow{\mathcal{D}}(s)_1\}$$
$$\overrightarrow{\mathcal{D}}([t,s])_2 \stackrel{\text{def}}{=} x \vdash \mathbf{case}\, x\, \mathbf{of}\, \{\mathbf{inr}\ x \to \mathbf{lfst} \,;\, \overrightarrow{\mathcal{D}}(t)_2 | x \to \mathbf{lsnd} \,;\, \overrightarrow{\mathcal{D}}(s)_2\}$$
$$.$$

Then, we define the following semantics:

$$\begin{aligned}
\{\!|\overrightarrow{\mathcal{D}}(\tau+\sigma)_1|\!\} &\stackrel{\text{def}}{=} \{\!|\overrightarrow{\mathcal{D}}(\tau)_1|\!\} + \{\!|\overrightarrow{\mathcal{D}}(\tau)_1|\!\} \\
\{\!|\overrightarrow{\mathcal{D}}(\tau+\sigma)_2|\!\} &\stackrel{\text{def}}{=} \{\!|\overrightarrow{\mathcal{D}}(\tau)_2|\!\} \times \{\!|\overrightarrow{\mathcal{D}}(\tau)_2|\!\} \\
\{\!|\overrightarrow{\mathcal{D}}(\mathbf{inl}\,)_1|\!\} &\stackrel{\text{def}}{=} \iota_1 \\
\{\!|\overrightarrow{\mathcal{D}}(\mathbf{inl}\,)_2|\!\} &\stackrel{\text{def}}{=} _ \mapsto x \mapsto (x,0) \\
\{\!|\overrightarrow{\mathcal{D}}(\mathbf{inr}\,)_1|\!\} &\stackrel{\text{def}}{=} \iota_2 \\
\{\!|\overrightarrow{\mathcal{D}}(\mathbf{inr}\,)_2|\!\} &\stackrel{\text{def}}{=} _ \mapsto y \mapsto (0,y) \\
\{\!|\overrightarrow{\mathcal{D}}([t,s])_1|\!\} &\stackrel{\text{def}}{=} [\{\!|\overrightarrow{\mathcal{D}}(t)_1|\!\}, \{\!|\overrightarrow{\mathcal{D}}(s)_1|\!\}] \\
\{\!|\overrightarrow{\mathcal{D}}([t,s])_2|\!\} &\stackrel{\text{def}}{=} [x \mapsto (x',_) \mapsto \{\!|\overrightarrow{\mathcal{D}}(t)_2|\!\}(x)(x'), y \mapsto (y',_) \mapsto \{\!|\overrightarrow{\mathcal{D}}(t)_2|\!\}(y)(y')] \\
&.
\end{aligned}$$

We define the forward AD logical relation $P_{\tau+\sigma}$ for coproducts on
$(\mathbb{R} \to (\{\!|\tau|\!\} + \{\!|\sigma|\!\})) \times ((\mathbb{R} \to (\{\!|\overrightarrow{\mathcal{D}}(\tau)_1|\!\} + \{\!|\overrightarrow{\mathcal{D}}(\sigma)_1|\!\})) \times (\mathbb{R} \to \mathbb{R} \multimap (\{\!|\overrightarrow{\mathcal{D}}(\tau)_2|\!\} \times \{\!|\overrightarrow{\mathcal{D}}(\sigma)_2|\!\})))$
as

$$\begin{aligned}
&\{(f';\iota_1,(g';\iota_1, x \mapsto x' \mapsto (h(x)(x'),0))) \mid (f',(g',h')) \in P_\tau\} \cup \\
&\{(f';\iota_2,(g';\iota_2, x \mapsto x' \mapsto (0,h(x)(x')))) \mid (f',(g',h')) \in P_\sigma\}\,.
\end{aligned}$$

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 $(f,(g,h)) \in P_{\tau+\sigma}$, $(\{\!|t|\!\},(\{\!|\overrightarrow{\mathcal{D}}(t)_1|\!\},\{\!|\overrightarrow{\mathcal{D}}(t)_2|\!\})) \in P_\tau$ and

$$(\{\!|s|\!\},(\{\!|\overrightarrow{\mathcal{D}}(s)_1|\!\},\{\!|\overrightarrow{\mathcal{D}}(s)_2|\!\})) \in P_\sigma.$$

We have to show that

$$\begin{aligned}
&(f;[\{\!|t|\!\},\{\!|s|\!\}], \\
&\qquad (g;[\{\!|\overrightarrow{\mathcal{D}}(t)_1|\!\},\{\!|\overrightarrow{\mathcal{D}}(s)_1|\!\}], \\
&\qquad\ z \mapsto z' \mapsto [x \mapsto (x',_) \mapsto \{\!|\overrightarrow{\mathcal{D}}(t)_2|\!\}(x)(x'),
\end{aligned}$$

$$y \mapsto (y', \_) \mapsto \{\!|\overrightarrow{\mathcal{D}}(t)_2|\!\}(y)(y')](g(z))(h(z)(z')))) \in P_{\{\!|\rho|\!\}}.$$

Now, we have two cases:

- $(f, (g, h)) = (f'; \iota_1, (g'; \iota_1, x \mapsto x' \mapsto (h'(x)(x'), 0)))$, for $(f', (g', h')) \in P_\tau$. Then,

$$\begin{aligned}&(f; [\{\!|t|\!\}, \{\!|s|\!\}],\\ &\quad (g; [\{\!|\overrightarrow{\mathcal{D}}(t)_1|\!\}, \{\!|\overrightarrow{\mathcal{D}}(s)_1|\!\}],\\ &\quad z \mapsto z' \mapsto [x \mapsto (x', \_) \mapsto \{\!|\overrightarrow{\mathcal{D}}(t)_2|\!\}(x)(x'),\\ &\qquad y \mapsto (y', \_) \mapsto \{\!|\overrightarrow{\mathcal{D}}(t)_2|\!\}(y)(y')](g(z))(h(z)(z')))) =\\ &(f'; \{\!|t|\!\}, (g'; \{\!|\overrightarrow{\mathcal{D}}(t)_1|\!\}, z \mapsto z' \mapsto \{\!|\overrightarrow{\mathcal{D}}(t)_2|\!\}(g(z))(h(z)(z')))),\end{aligned}$$

  which is a member of P_ρ because t respects the logical relation by assumption.
- $(f, (g, h)) = (f'; \iota_2, (g'; \iota_2, x \mapsto x' \mapsto (0, h'(x)(x'))))$ for $(f', (g', h')) \in P_\sigma$. Then,

$$\begin{aligned}&(f; [\{\!|t|\!\}, \{\!|s|\!\}],\\ &\quad (g; [\{\!|\overrightarrow{\mathcal{D}}(t)_1|\!\}, \{\!|\overrightarrow{\mathcal{D}}(s)_1|\!\}],\\ &\quad z \mapsto z' \mapsto [x \mapsto (x', \_) \mapsto \{\!|\overrightarrow{\mathcal{D}}(t)_2|\!\}(x)(x'),\\ &\qquad y \mapsto (y', \_) \mapsto \{\!|\overrightarrow{\mathcal{D}}(t)_2|\!\}(y)(y')](g'(z))(h'(z)(z')))) =\\ &(f'; \{\!|s|\!\}, (g'; \{\!|\overrightarrow{\mathcal{D}}(s)_1|\!\}, z \mapsto z' \mapsto \{\!|\overrightarrow{\mathcal{D}}(t)_2|\!\}(g'(z))(h'(z)(z')))),\end{aligned}$$

  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.

Reverse AD. We have that

$$\begin{aligned}&\overleftarrow{\mathcal{D}}(\mathbf{inl}\,)_1 \in \mathbf{ALSyn}(\overleftarrow{\mathcal{D}}(\tau)_1, \overleftarrow{\mathcal{D}}(\tau)_1 + \overleftarrow{\mathcal{D}}(\tau)_2)\\ &\overleftarrow{\mathcal{D}}(\mathbf{inl}\,)_2 \in \mathbf{ALSyn}(\overleftarrow{\mathcal{D}}(\tau)_1, \mathbf{LFun}(\overleftarrow{\mathcal{D}}(\tau)_2 * \overleftarrow{\mathcal{D}}(\sigma)_2, \overleftarrow{\mathcal{D}}(\tau)_2))\\ &\overleftarrow{\mathcal{D}}(\mathbf{inr}\,)_1 \in \mathbf{ALSyn}(\overleftarrow{\mathcal{D}}(\sigma)_1, \overleftarrow{\mathcal{D}}(\tau)_1 + \overleftarrow{\mathcal{D}}(\tau)_2)\\ &\overleftarrow{\mathcal{D}}(\mathbf{inr}\,)_2 \in \mathbf{ALSyn}(\overleftarrow{\mathcal{D}}(\sigma)_1, \mathbf{LFun}(\overleftarrow{\mathcal{D}}(\tau)_2 * \overleftarrow{\mathcal{D}}(\sigma)_2, \overleftarrow{\mathcal{D}}(\sigma)_2))\\ &\overleftarrow{\mathcal{D}}([t, s])_1 \in \mathbf{ALSyn}(\overleftarrow{\mathcal{D}}(\tau)_1 + \overleftarrow{\mathcal{D}}(\sigma)_1, \overleftarrow{\mathcal{D}}(\rho)_1)\\ &\overleftarrow{\mathcal{D}}([t, s])_2 \in \mathbf{ALSyn}(\overleftarrow{\mathcal{D}}(\tau)_1 + \overleftarrow{\mathcal{D}}(\sigma)_1, \mathbf{LFun}(\overleftarrow{\mathcal{D}}(\rho)_2, \overleftarrow{\mathcal{D}}(\tau)_2 * \overleftarrow{\mathcal{D}}(\sigma)_2)).\end{aligned}$$

We can define

$$
\begin{aligned}
\overleftarrow{\mathcal{D}}(\tau+\sigma)_1 &\stackrel{\text{def}}{=} \overleftarrow{\mathcal{D}}(\tau)_1 + \overleftarrow{\mathcal{D}}(\sigma)_1\\
\overleftarrow{\mathcal{D}}(\tau+\sigma)_2 &\stackrel{\text{def}}{=} \overleftarrow{\mathcal{D}}(\tau)_1 * \overleftarrow{\mathcal{D}}(\sigma)_1\\
\overleftarrow{\mathcal{D}}(\mathbf{inl}\,)_1 &\stackrel{\text{def}}{=} \mathbf{inl}\\
\overleftarrow{\mathcal{D}}(\mathbf{inl}\,)_2 &\stackrel{\text{def}}{=} \lambda\_.\mathbf{lfst}\\
\overleftarrow{\mathcal{D}}(\mathbf{inr}\,)_1 &\stackrel{\text{def}}{=} \mathbf{inr}\\
\overleftarrow{\mathcal{D}}(\mathbf{inr}\,)_2 &\stackrel{\text{def}}{=} \lambda\_.\mathbf{lsnd}\\
\overleftarrow{\mathcal{D}}([t,s])_1 &\stackrel{\text{def}}{=} x \vdash \mathbf{case}\, x\, \mathbf{of}\, \{\mathbf{inl}\ x \to \overleftarrow{\mathcal{D}}(t)_1 | x \to \overleftarrow{\mathcal{D}}(s)_1\}\\
\overleftarrow{\mathcal{D}}([t,s])_2 &\stackrel{\text{def}}{=} x \vdash \mathbf{case}\, x\, \mathbf{of}\, \{\mathbf{inr}\ x\ \to \mathbf{lpair}(\overleftarrow{\mathcal{D}}(t)_2, \underline{0}) | x \to \mathbf{lpair}(\underline{0}, \overleftarrow{\mathcal{D}}(s)_2)\}\\
&.
\end{aligned}
$$

Then,

$$
\begin{aligned}
\{\!|\overleftarrow{\mathcal{D}}(\tau+\sigma)_1|\!\} &\stackrel{\text{def}}{=} \{\!|\overleftarrow{\mathcal{D}}(\tau)_1|\!\} + \{\!|\overleftarrow{\mathcal{D}}(\tau)_1|\!\}\\
\{\!|\overleftarrow{\mathcal{D}}(\tau+\sigma)_2|\!\} &\stackrel{\text{def}}{=} \{\!|\overleftarrow{\mathcal{D}}(\tau)_2|\!\} \times \{\!|\overleftarrow{\mathcal{D}}(\tau)_2|\!\}\\
\{\!|\overleftarrow{\mathcal{D}}(\mathbf{inl}\,)_1|\!\} &\stackrel{\text{def}}{=} \iota_1\\
\{\!|\overleftarrow{\mathcal{D}}(\mathbf{inl}\,)_2|\!\} &\stackrel{\text{def}}{=} \_ \mapsto (x, \_) \mapsto x\\
\{\!|\overleftarrow{\mathcal{D}}(\mathbf{inr}\,)_1|\!\} &\stackrel{\text{def}}{=} \iota_2\\
\{\!|\overleftarrow{\mathcal{D}}(\mathbf{inr}\,)_2|\!\} &\stackrel{\text{def}}{=} \_ \mapsto (\_, y) \mapsto y\\
\{\!|\overleftarrow{\mathcal{D}}([t,s])_1|\!\} &\stackrel{\text{def}}{=} [\{\!|\overleftarrow{\mathcal{D}}(t)_1|\!\}, \{\!|\overleftarrow{\mathcal{D}}(s)_1|\!\}]\\
\{\!|\overleftarrow{\mathcal{D}}([t,s])_2|\!\} &\stackrel{\text{def}}{=} [x \mapsto z' \mapsto (\{\!|\overleftarrow{\mathcal{D}}(t)_2|\!\}(x)(z'), 0), y \mapsto z' \mapsto (0, \{\!|\overleftarrow{\mathcal{D}}(t)_2|\!\}(y)(z'))]\\
&.
\end{aligned}
$$

We define the reverse AD logical relation $P_{\tau+\sigma}$ for coproducts on
$(\mathbb{R} \to (\{\!|\tau|\!\} + \{\!|\sigma|\!\})) \times ((\mathbb{R} \to (\{\!|\overleftarrow{\mathcal{D}}(\tau)_1|\!\} + \{\!|\overleftarrow{\mathcal{D}}(\sigma)_1|\!\})) \times (\mathbb{R} \to (\{\!|\overleftarrow{\mathcal{D}}(\tau)_2|\!\} \times \{\!|\overleftarrow{\mathcal{D}}(\sigma)_2|\!\}) \multimap \mathbb{R}))$
as

$$
\begin{aligned}
&\{(f';\iota_1, (g';\iota_1, z \mapsto (x', \_) \mapsto h'(z)(x'))) \mid (f', (g', h')) \in P_\tau\} \cup\\
&\{(f';\iota_2, (g';\iota_2, z \mapsto (\_, y') \mapsto h'(z)(y'))) \mid (f', (g', h')) \in P_\sigma\}.
\end{aligned}
$$

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 $(f, (g, h)) \in P_{\tau+\sigma}$, $(\{\!|t|\!\}, (\{\!|\overleftarrow{\mathcal{D}}(t)_1|\!\}, \{\!|\overleftarrow{\mathcal{D}}(t)_2|\!\})) \in P_\tau$ and

$$(\{\!|s|\!\}, (\{\!|\overleftarrow{\mathcal{D}}(s)_1|\!\}, \{\!|\overleftarrow{\mathcal{D}}(s)_2|\!\})) \in P_\sigma.$$

We have to show that

$$\begin{aligned}
&(f; [\{\!|t|\!\}, \{\!|s|\!\}],\\
&\quad (g; [\{\!|\overleftarrow{\mathcal{D}}(t)_1|\!\}, \{\!|\overleftarrow{\mathcal{D}}(s)_1|\!\}],\\
&\quad z \mapsto x' \mapsto h(z)([x \mapsto z' \mapsto (\{\!|\overleftarrow{\mathcal{D}}(t)_2|\!\}(x)(z'), 0),\\
&\qquad\qquad y \mapsto z' \mapsto (0, \{\!|\overleftarrow{\mathcal{D}}(s)_2|\!\}(y)(z'))](g(x))(x')))) \in P_{\{\!|\rho|\!\}}.
\end{aligned}$$

Now, we have two cases:

- $(f, (g, h)) = (f'; \iota_1, (g'; \iota_1, z \mapsto (x', \_) \mapsto h'(z)(x')))$, for $(f', (g', h')) \in P_\tau$. Then,

$$\begin{aligned}
&(f; [\{\!|t|\!\}, \{\!|s|\!\}],\\
&\quad (g; [\{\!|\overleftarrow{\mathcal{D}}(t)_1|\!\}, \{\!|\overleftarrow{\mathcal{D}}(s)_1|\!\}],\\
&\quad z \mapsto x' \mapsto h(z)([x \mapsto z' \mapsto (\{\!|\overleftarrow{\mathcal{D}}(t)_2|\!\}(x)(z'), 0),\\
&\qquad\qquad y \mapsto z' \mapsto (0, \{\!|\overleftarrow{\mathcal{D}}(s)_2|\!\}(y)(z'))](g(x))(x')))) =\\
&(f'; \{\!|t|\!\}, (g'; \{\!|\overleftarrow{\mathcal{D}}(t)_1|\!\}, z \mapsto x' \mapsto h'(z)(\{\!|\overleftarrow{\mathcal{D}}(t)_2|\!\}(g'(x))(x')))),
\end{aligned}$$

  which is a member of P_ρ because t respects the logical relation by assumption.
- $(f, (g, h)) = (f'; \iota_2, (g'; \iota_2, z \mapsto (\_, y') \mapsto h'(z)(y')))$ for $(f', (g', h')) \in P_\sigma$. Then,

$$\begin{aligned}
&(f; [\{\!|t|\!\}, \{\!|s|\!\}],\\
&\quad (g; [\{\!|\overleftarrow{\mathcal{D}}(t)_1|\!\}, \{\!|\overleftarrow{\mathcal{D}}(s)_1|\!\}],\\
&\quad z \mapsto x' \mapsto h(z)([x \mapsto z' \mapsto (\{\!|\overleftarrow{\mathcal{D}}(t)_2|\!\}(x)(z'), 0),\\
&\qquad\qquad y \mapsto z' \mapsto (0, \{\!|\overleftarrow{\mathcal{D}}(s)_2|\!\}(y)(z'))](g(x))(x')))) =\\
&(f'; \{\!|s|\!\}, (g'; \{\!|\overleftarrow{\mathcal{D}}(s)_1|\!\}, z \mapsto x' \mapsto h'(z)(\{\!|\overleftarrow{\mathcal{D}}(s)_2|\!\}(g'(x))(x')))),
\end{aligned}$$

  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

$$(A_1, A_2) + (B_1, B_2) \stackrel{\text{def}}{=} (A_1 + B_1, A_2 \times B_2)$$

lifts the coproduct in $\mathcal{C}$ to a *weak* (fibred) 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).

Acknowledgments. We thank Michael Betancourt, Bob Carpenter, Mathieu Huot, Ohad Kammar, Gabriele Keller, Gordon Plotkin, Curtis Chin Jen Sem, Amir Shaikhha, Tom Smeding, and Sam Staton for helpful discussions about Automatic Differentiation.

References

[1] **Abadi, M. and Plotkin, G. D.** 2020. A simple differentiable programming language. In *Proc. POPL 2020.* ACM.

[2] **Adámek, J. and Koubek, V.** 1979. Least fixed point of a functor. *Journal of Computer and System Sciences*, 19(2):163–178.

[3] **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*, pp. 36–54. Springer.

[4] **Alvarez-Picallo, M.**, **Ghica, D. R.**, **Sprunger, D.**, **and Zanasi, F.** 2021. Functorial string diagrams for reverse-mode automatic differentiation. *arXiv preprint arXiv:2107.13433.*

[5] **Barr, M.** 1993. Terminal coalgebras in well-founded set theory. *Theoret. Comput. Sci.*, 114(2):299–315.

[6] **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 *Proc. ESOP 2020.* Springer. To appear.

[7] **Baydin, A. G.**, **Pearlmutter, B. A.**, **Radul, A. A.**, **and Siskind, J. M.** 2018. Automatic differentiation in machine learning: a survey. *Journal of machine learning research*, 18.

[8] **Boman, J.** 1968. Differentiability of a function and of its compositions with functions of one variable. *Mathematica Scandinavica*, pp. 249–268.

[9] **Brunel, A.**, **Mazza, D.**, **and Pagani, M.** 2020. Backpropagation in the simply typed lambda-calculus with linear negation. In *Proc. POPL 2020.*

[10] **Carboni, A.**, **Lack, S.**, **and Walters, R. F. C.** 1993. Introduction to extensive and distributive categories. *J. Pure Appl. Algebra*, 84(2):145–158.

[11] **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 *Proc. CSL 2020.*

[12] **Dubuc, E.** 1968. Adjoint triangles. In *Reports of the Midwest Category Seminar, II*, pp. 69–91. Springer, Berlin.

[13] **Elliott, C.** 2018. The simple essence of automatic differentiation. *Proceedings of the ACM on Programming Languages*, 2(ICFP):70.

[14] **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)*, pp. 1–13. IEEE.
[15] **Gray, J. W.** 1966. Fibred and cofibred categories. In *Proc. Conf. Categorical Algebra (La Jolla, Calif., 1965)*, pp. 21–83. Springer, New York.
[16] **Griewank, A. and Walther, A.** 2008. *Evaluating derivatives: principles and techniques of algorithmic differentiation*, volume 105. Siam.
[17] **Huot, M.**, **Staton, S.**, **and Vákár, M.** 2021. Higher order automatic differentiation of higher order functions. *arXiv preprint arXiv:2101.06757.*
[18] **Huot, M.**, **Staton, S.**, **and Vákár, M.** 2020. Correctness of automatic differentiation via diffeologies and categorical gluing. In *Proc. FoSSaCS.*
[19] **Jacobs, B.** 1999. *Categorical logic and type theory.* Elsevier.
[20] **Johnstone, P. T.** 2002. *Sketches of an elephant: A topos theory compendium*, volume 2. Oxford University Press.
[21] **Kelly, G. M.** 1974. Doctrinal adjunction. In *Category Seminar (Proc. Sem., Sydney, 1972/1973)*, pp. 257–280. Lecture Notes in Math., Vol. 420.
[22] **Krawiec, F.**, **Krishnaswami, N.**, **Peyton Jones, S.**, **Ellis, T.**, **Fitzgibbon, A.**, **and Eisenberg, R.** 2021. Provably correct, asymptotically efficient, higher-order reverse-mode automatic differentiation. https://www.microsoft.com/en-us/research/uploads/prod/2021/08/higher-order-ad.pdf.
[23] **Lack, S.** 2012. Non-canonical isomorphisms. *J. Pure Appl. Algebra*, 216(3):593–597.
[24] **Lambek, J. and Scott, P. J.** 1988. *Introduction to higher-order categorical logic*, volume 7. Cambridge University Press.
[25] **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*, pp. 6–7.
[26] **Lucatelli Nunes, F.** 2016. On biadjoint triangles. *Theory Appl. Categ.*, 31:Paper No. 9, 217–256.
[27] **Lucatelli Nunes, F.** 2017. Pseudomonads and Descent, PhD Thesis (Chapter 1). *University of Coimbra.* arXiv: 1802.01767.
[28] **Lucatelli Nunes, F.** 2019. Pseudoalgebras and non-canonical isomorphisms. *Appl. Categ. Structures*, 27(1):55–63.
[29] **Lucatelli Nunes, F.** 2021. Descent data and absolute Kan extensions. *Theory Appl. Categ.*, 37:Paper No. 18, 530–561.
[30] **MacDonald, J. and Sobral, M.** 2004. Aspects of monads. In *Categorical foundations*, volume 97 of *Encyclopedia Math. Appl.*, pp. 213–268. Cambridge Univ. Press, Cambridge.
[31] **Mak, C. and Ong, L.** 2020. A differential-form pullback programming language for higher-order reverse-mode automatic differentiation. arxiv:2002.08241.
[32] **Mazza, D. and Pagani, M.** 2021. Automatic differentiation in pcf. *Proceedings of the ACM on Programming Languages*, 5(POPL):1–27.
[33] **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.
[34] **Pitts, A. M.** 1995. Categorical logic. Technical report, University of Cambridge, Computer Laboratory.
[35] **Plotkin, G.** 2018. Some principles of differential programming languages. *Invited talk, POPL*, 2018.

[36] **Santocanale, L.** 2002. μ-bicomplete categories and parity games. *RAIRO - Theoretical Informatics and Applications - Informatique Théorique et Applications*, 36(2):195–227.

[37] **Speelpenning, B.** 1980. Compiling fast partial derivatives of functions given by algorithms. Technical report, Illinois Univ., Urbana (USA). Dept. of Computer Science.

[38] **Vákár, M.** 2017. In search of effectful dependent types. *arXiv preprint arXiv:1706.07997*. DPhil Thesis, University of Oxford.

[39] **Vákár, M.** 2021. Reverse AD at higher types: Pure, principled and denotationally correct. In *ESOP*, pp. 607–634.

[40] **Vákár, M. and Smeding, T.** 2021. CHAD: Combinatory homomorphic automatic differentiation. *arXiv preprint arXiv:2103.15776*.

[41] **Vytiniotis, D.**, **Belov, D.**, **Wei, R.**, **Plotkin, G.**, **and Abadi, M.** 2019. The differentiable curry.

[42] **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).

[43] **Wengert, R. E.** 1964. A simple automatic derivative evaluation program. *Communications of the ACM*, 7(8):463–464.

Fernando Lucatelli Nunes
Department of Information and Computing Sciences, Utrecht University, Netherlands
& CMUC, Centre for Mathematics, University of Coimbra, Portugal

E-mail address: `f.lucatellinunes@uu.nl`

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

E-mail address: `m.i.l.vakar@uu.nl`