induced by coprojections and projections, provided that

$$H : \left(\mathbf{Syn}_V, \mathbf{Syn}_{\mathcal{S}}, \underline{\nu}_{\mathbf{Syn}}\right) \to \mathcal{U}_{r\mathcal{BV}}\left(\mathcal{V}, \mathcal{T}\right)$$

is an rCBV model morphism and $(\mathcal{V}, \mathcal{T})$ *is an rCBV* $\boldsymbol{\omega}$**Cpo**-*pair.*

Fernando Lucatelli Nunes
Departement Informatica, Universiteit Utrecht, Nederland & University of Coimbra, CMUC, Department of Mathematics, Portugal
E-mail address: f.lucatellinunes@uu.nl

Matthijs Vákár
Departement Informatica, Universiteit Utrecht, Nederland
E-mail address: m.i.l.vakar@uu.nl

Pré-Publicações do Departamento de Matemática
Universidade de Coimbra
Preprint Number 22–31

LOGICAL RELATIONS FOR PARTIAL FEATURES AND AUTOMATIC DIFFERENTIATION CORRECTNESS

FERNANDO LUCATELLI NUNES AND MATTHIJS VÁKÁR

Abstract: We present a simple technique for semantic, open logical relations arguments about languages with recursive types, which, as we show, follows from a principled foundation in categorical semantics. We demonstrate how it can be used to give a very straightforward proof of correctness of practical forward- and reverse-mode dual numbers style automatic differentiation (AD) on ML-family languages. The key idea is to combine it with a suitable open logical relations technique for reasoning about differentiable partial functions (a suitable lifting of the partiality monad to logical relations), which we introduce.

Keywords: Recursive Types, Logical relations, Automatic differentiation, Programming Languages, Functional Programming, Denotational Semantics, Recursion.
Math. Subject Classification (2020): 68N15, 68N18, 68Q55, 68W30, 18D20, 18A25, 18C15.

Introduction

Automatic differentiation (AD) computes derivatives in a numerically stable way that scales efficiently to high-dimensional spaces. Its ubiquity in scientific computing, statistics and machine learning applications has led to the idea of differentiable programming: compilers for modern programming languages should provide good built-in support for AD of any program written in those languages [22, 26]. It is a non-trivial question how to efficiently and correctly differentiate arbitrary complex programs, which has led to a booming area of research.

Dual numbers techniques give very simple forward and reverse mode AD algorithms for expressive ML-family functional languages [27, 6, 13, 21, 16, 29]. The correctness proofs for these algorithms rely on (open) semantic logical relations

Received October 21, 2022.
This project has received funding via NWO Veni grant number VI.Veni.202.124 as well as the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No. 895827. This research was supported through the programme "Oberwolfach Leibniz Fellows" by the Mathematisches Forschungsinstitut Oberwolfach in 2022. It was also partially supported by the CMUC, Centre for Mathematics of the University of Coimbra - UIDB/00324/2020, funded by the Portuguese Government through FCT/MCTES.

arguments. These proofs become surprisingly complex, even for simple dual numbers AD algorithms, when considering AD of languages with partial features such as recursion. For example, the authors have given such proofs in the past (in the preprint [33]), but despite our best efforts they seemed to require subtle combinations of sheaves (to deal with the surprisingly hard interaction between partiality and differentiation induced by conditionals on real numbers) and ω-cpo-structure (required for recursion). The resulting proofs were dissatisfying due to their complexity, restricting the audience and obstructing their generalisation to more advanced AD algorithms like CHAD [35, 19].

The present work is a vast simplification of the arguments in [33] (and, also, [13]), not requiring any (ω-cpos internal to) sheaves or diffeological spaces and instead relying on plain ω-cpos. This simplification is desirable if we are to apply the techniques more generally, such as to the correctness of CHAD [35, 19] for recursion, and if we want the technique to be more broadly accessible.

The first contribution of this paper lies in the development of very simple but powerful, semantic logical relations techniques for reasoning about recursive types. By contrast with the existing techniques of [25, 2], our technique follows directly from standard category theoretic recipes and applies equally to open logical relations (sconing along functors G other than $\mathrm{Hom}\,(1,-)$) [3].

The second contribution is the development of a simple logical relations technique for reasoning about partially defined differentiable functions, which can be understood as a particular lifting of the partiality monad to our logical relations.

Thirdly, we show that the combination of these two techniques suffices to give a very elegant correctness proof of the practically useful dual-numbers style AD algorithms of expressive ML-family languages implemented by [29].

Finally, our arguments can be generalised to the correctness argument of the more advanced AD technique CHAD when applied to languages with partial features, which motivates a lot of this development.

1. Why we care about differentiating partial programs?

Given the central role that AD plays in modern scientific computing and machine learning, the ideal of differential programming is emerging [22, 26]: compilers for general purpose programming languages should provide built-in support for automatic differentiation of any programs written in the language. What a correct and efficient notion of derivative is of some popular programming language features might not be so straightforward, however, as they often

We define the set $\mathfrak{P}^{\mathfrak{d}0}_{\mathsf{T}_p}(\mathcal{V},\mathcal{T},\underline{\nu})$ *of* positive types *over* T_p *by* (C.4)*, that is to say, the subset of the* 0*-variable parametric types in* $\mathfrak{P}^{\mathfrak{d}}_{\mathsf{T}_p}(\mathcal{V},\mathcal{T},\underline{\nu})$*.*

$$\times \circ (G \times G') , \sqcup \circ (G \times G') : (\mathcal{V}^{\mathrm{op}} \times \mathcal{V})^{2n} \to \mathcal{V} \quad \text{(C.1)} \qquad G \circ \mathrm{diag}_n : \mathcal{V}^{\mathrm{op}} \times \mathcal{V} \to \mathcal{V} \quad \text{(C.2)}$$

$$G, G' : (\mathcal{V}^{\mathrm{op}} \times \mathcal{V})^{n} \to \mathcal{V} \quad \text{(C.3)} \qquad \left\{ A \in \mathcal{V} : A \in \mathfrak{P}^{\mathfrak{d}}_{\mathsf{T}_p}(\mathcal{V},\mathcal{T},\underline{\nu}) \right\} \quad \text{(C.4)}$$

In the case of syntactic $rCBV$ models (see 2.3), taking T_p to be the set of primitive types, the definition above provides us with a formal definition of data types in our context. By the universal property of these syntactic $rCBV$ models, we can establish the semantics and the logical relations as $rCBV$ model morphisms (as we do in 5.1 and 6). Hence, it is particularly useful to understand the image of (recursive) data types by $rCBV$ model morphisms. The result below shows that the image of such a data type is always the coproduct of finite products of the image of the primitive types by the $rCBV$ model.

Theorem C.2. *Let* $(\mathcal{V},\mathcal{T},\underline{\nu})$ *be an* $rCBV$ *model, and* T_p *a finite set of objects of* $\mathcal{V}$*. For each* $D \in \mathfrak{P}^{\mathfrak{d}0}_{\mathsf{T}_p}(\mathcal{V},\mathcal{T},\underline{\nu})$*, there is a countable family*

$$\left(\mathsf{m}_{(j,K)} \in \mathbb{N}\right)_{(j,K)\in L\times \mathsf{T}_p}$$

such that, for any $rCBV$ *model morphism* $H : (\mathcal{V},\mathcal{T}) \to \mathcal{U}_{r\mathcal{BV}}(\mathcal{V}',\mathcal{T}')$*,*

$$H(D) \cong \coprod_{j\in L} \left(\prod_{K\in \mathsf{T}_p} H(K)^{\mathsf{m}_{(j,K)}} \right),$$

where the isomorphism $\cong$ *is induced by coprojections and projections.*[‖]

Let $\left(\mathbf{Syn}_V, \mathbf{Syn}_{\mathcal{S}}, \underline{\nu}_{\mathbf{Syn}}\right)$ be the syntactic $rCBV$ model established in Section 4.3. The positive types over **real** are precisely those types corresponding to data types in our source language. Therefore:

Theorem C.3. *Let* $(\mathcal{V},\mathcal{T})$ *be an* $rCBV$ $\boldsymbol{\omega}$**Cpo**-*pair. For each* $\tau \in \mathbf{Syn}_V$ *corresponding to a (possibly recursive) data type, there is a countable family* $(s_r \in \mathbb{N})_{r\in\mathfrak{L}}$ *such that there is an isomorphism*

$$H(\tau) \cong \coprod_{r\in s_r} H(\mathbf{real})^{s_r},$$

[‖] $\cong$ is induced by the universal property – in other words, it is just a reorganization of the involved coproducts and products.

$$\mathbf{let}\, x = v\, \mathbf{in}\, t = t[^{v}/_{x}] \qquad \mathbf{let}\, y = (\mathbf{let}\, x = t\, \mathbf{in}\, s)\, \mathbf{in}\, r = \mathbf{let}\, x = t\, \mathbf{in}\, (\mathbf{let}\, y = s\, \mathbf{in}\, r)$$

$$\mathbf{case}\, \mathbf{inl}\, v\, \mathbf{of}\, \{\mathbf{inl}\, x \to t \mid \mathbf{inr}\, y \to s\} = t[^{v}/_{x}] \qquad t[^{v}/_{z}] \overset{\#x,y}{=} \mathbf{case}\, v\, \mathbf{of}\, \{\begin{array}{l}\mathbf{inl}\, x \to t[^{\mathbf{inl}\, x}/_{z}] \\ \mid \mathbf{inr}\, y \to t[^{\mathbf{inr}\, y}/_{z}]\end{array}\}$$

$$\mathbf{case}\, \mathbf{inr}\, v\, \mathbf{of}\, \{\mathbf{inl}\, x \to t \mid \mathbf{inr}\, y \to s\} = s[^{v}/_{y}]$$

$$\mathbf{case}\, \langle v, w\rangle\, \mathbf{of}\, \langle x, y\rangle \to t = t[^{v}/_{x}, {}^{w}/_{y}] \qquad t[^{v}/_{z}] \overset{\#x,y}{=} \mathbf{case}\, v\, \mathbf{of}\, \langle x, y\rangle \to t[^{\langle x,y\rangle}/_{z}]$$

$$(\lambda x.t)\; v = t[^{v}/_{x}] \qquad v \overset{\#x}{=} \lambda x.v\, x$$

$$\mathbf{case}\, \mathbf{roll}\, v\, \mathbf{of}\, \mathbf{roll}\, x \to t = t[^{v}/_{x}] \qquad t[^{v}/_{z}] \overset{\#x}{=} \mathbf{case}\, v\, \mathbf{of}\, \mathbf{roll}\, x \to t[^{\mathbf{roll}\, x}/_{z}]$$

FIGURE B.3. The standard $\beta\eta$-equational theory for a *CBV* language with recursive types. We write $\overset{\#x_1,\ldots,x_n}{=}$ to indicate that the variables $x_1, \ldots, x_n$ are fresh in the left hand side. In the second rule, x may not be free in r. Equations hold on pairs of terms of the same type.

Appendix C. Image of recursive types

While the logical relations for the primitive types are the primitive ones (defined by the image of each primitive object by $[\![-]\!]_\tau$), we get the logical relations of more general data types out of the fact that $[\![-]\!]_\tau$ is structure preserving. We finish this section establishing the result that underlies the computation of the logical relations for data types in our setting.

We start by giving the definition corresponding to data/positive types for any *rCBV* model with a chosen *finite* set T_p of objects, playing the role of the set of primitive types. These are the objects inductively defined by finite products, finite coproducts and recursion of objects in T_p.

Definition C.1. Let $(\mathcal{V}, \mathcal{T}, \underline{\nu})$ be an *rCBV* model, and T_p a finite set of objects of $\mathcal{V}$. For each $K \in \mathsf{T}_p$, let $\underline{\mathsf{K}}, I, O : \mathcal{V}^{\mathrm{op}} \times \mathcal{V} \to \mathcal{V}$ be the constant functors which are, respectively, equal to $\underline{\mathsf{K}}$, $\mathsf{1}$ and $\mathsf{0}$. *We define the set* $\mathfrak{P}^{\mathfrak{d}}_{\mathsf{T}_p}(\mathcal{V}, \mathcal{T}, \underline{\nu})$ *inductively by (D1), (D2) and (D3).*

(D1) The functors $\underline{\mathsf{K}}, I, O$ are in $\mathfrak{P}^{\mathfrak{d}}_{\mathsf{T}_p}(\mathcal{V}, \mathcal{T}, \underline{\nu})$. Moreover, the projection $\pi_2 : \mathcal{V}^{\mathrm{op}} \times \mathcal{V} \to \mathcal{V}$ belongs to $\mathfrak{P}^{\mathfrak{d}}_{\mathsf{T}_p}(\mathcal{V}, \mathcal{T}, \underline{\nu})$.

(D2) For each $n \in \mathbb{N}^*$, if the functors (C.3) belong to $\mathfrak{P}^{\mathfrak{d}}_{\mathsf{T}_p}(\mathcal{V}, \mathcal{T}, \underline{\nu})$, then the functors (C.2) and (C.1) are in $\mathfrak{P}^{\mathfrak{d}}_{\mathsf{T}_p}(\mathcal{V}, \mathcal{T}, \underline{\nu})$.

(D3) If $E = (E_{\mathcal{V}}, E_{\mathcal{C}}) \in \mathfrak{P}^{\mathfrak{d}}_{\mathsf{T}_p}(\mathcal{V}, \mathcal{T}, \underline{\nu})$ is such that $E_{\mathcal{V}} \in \mathfrak{P}^{\mathfrak{d}}_{\mathsf{T}_p}(\mathcal{V}, \mathcal{T}, \underline{\nu})$, then $(\nu E_{\mathcal{V}})$ is in $\mathfrak{P}^{\mathfrak{d}}_{\mathsf{T}_p}(\mathcal{V}, \mathcal{T}, \underline{\nu})$.

go beyond what is studied in traditional calculus. In this paper we focus on the challenge posed, in particular, by partial language features: partial primitive operations, lazy conditionals on real numbers, iteration, recursion and recursive types.

Partial primitive operations are key: even the basic operations of division and logarithm are examples. (Lazy) conditionals on real numbers are used to paste together existing smooth functions, as basic example being the ReLU function

$$ReLU(x) \overset{\mathrm{def}}{=} \mathbf{if}\, x\, \mathbf{then}\, 0\, \mathbf{else}\, x\; = \mathbf{case}\, (\mathbf{sign}\; x)\, \mathbf{of}\, \{\mathbf{inl}\; _ \to 0 \mid \mathbf{inr}\; _ \to x\},$$

which is a key component of many neural networks. They are also frequently used in probabilistic programming to paste together density functions of different distributions [5]. People have long studied the subtle issue of how one should algorithmically differentiate such functions with "kinks" under the name of *the if-problem in automatic differentiation* [4]. Our solution is the one also employed by [1]: to treat the functions as semantically undefined at their kinks (at $x = 0$ in the case of $ReLU(x)$). This is justified given how coarse the semantic treatment of floating point numbers as real numbers is already. Our semantics based on partial functions defined on real numbers is sufficient to prove many high-level correctness properties. However, like any semantics based on real numbers, it fails to capture many of the low-level subtleties introduced by the floating point implementation. Our key insight that we use to prove correctness of AD of partial programs is to construct a suitable lifting of the partiality monad to a variant of [13]'s category of $\mathbb{R}^k$-indexed logical relations used to relate programs to their derivatives. This particular monad lifting for derivatives of partial functions can be seen as our solution to the if-problem in AD.

Similarly, iteration constructs, or while-loops, are necessary for implementing iterative algorithms with dynamic stopping criteria. Such algorithms are frequently used in programs that AD is applied to. For example, AD is applied to iterative differential equation solvers to perform Bayesian inference in SIR models. This technique played a key role in modelling the Covid19-pandemic [11]. For similar reasons, AD through iterative differential equation solvers is important for probabilistic modelling of pharmacokinetics [32]. Other common use-cases of iterative algorithms that need to be AD'ed are eigen-decompositions and algebraic equation solvers, such as those employed in Stan [7]. Finally, iteration gives a convenient way of achieving numerically stable approximations to complex functions (such as the Conway-Maxwell-Poisson density function [12]).

The idea is to construct, using iteration, a Taylor approximation that terminates once the next term in the series causes floating-point underflow. Indeed, for a function whose i-th terms in the Taylor expansion can be represented by a program

$$i : \mathbf{int}, x : \mathbf{real} \vdash t(i, x) : \mathbf{real},$$

we would define the underflow-truncated Taylor series by

$$\mathbf{iterate}\left(\begin{array}{l}\mathbf{case}\, x\, \mathbf{of}\, \langle x_1, x_2\rangle \to \mathbf{let}\, y = \, t(x_1, x_2)\, \mathbf{in} \\ \mathbf{case}\, -c < y < c\, \mathbf{of}\, \{\mathbf{inl}\, _ \to \mathbf{inr}\, x_2 \\ \qquad\qquad \mid \mathbf{inr}\, _ \to \mathbf{inl}\, \langle x_1 + 1, x_2 + y\rangle\}\end{array}\right) \mathbf{from}\, x = \langle 0, 0\rangle,$$

where c is a cut-off for floating-point underflow.

Next, recursive neural networks [31] are often mentioned as a use case of AD applied to recursive programs. While basic Child-Sum Tree-LSTMs can also be implemented with primitive recursion (a fold) over an inductively defined tree (which can be defined as a recursive type), there are other related models such as Top-Down-Tree-LSTMs that require an iterative or general recursive approach [36]. In fact, [15] has shown that a recursive approach is preferable as it better exposes the available parallelism in the model. In the extended version of this paper [20], we show some Haskell code for the recursive neural network of [30], to give an idea of how iteration and recursive types (in the form of inductive types of labelled trees) naturally arise in a functional implementation of such neural net architectures. We imagine that many more applications of AD applied to recursive programs with naturally emerge as the technique made available to machine learning researchers and engineers. Finally, we speculate that coinductive types like streams of real numbers, which can be encoded using recursive types as $\mu\alpha.\mathbf{1} \to (\mathbf{real} * \alpha)$, provide a useful API for on-line machine learning applications [28], where data is processed in real time as it becomes available.

2. Categorical models for languages with recursive types

We assume familiarity with basic category theory (see, for instance, [8]). We establish a class of categorical models for call-by-value (CBV) languages with tuple, variant, function, and recursive types, which we call *rCBV models*.

The first step is to establish the categorical model of computational λ_C-calculus (see [23]). This means that the underlying structure is that of a *Freyd-category*, see [18]. Notwithstanding that, we do not need to consider this level of

Appendix B. Typing rules for the target language

$$\frac{((x:\tau)\in\Gamma)}{\Delta \mid \Gamma \vdash x : \tau} \qquad \frac{\Delta \mid \Gamma \vdash t : \sigma \quad \Delta \mid \Gamma, x : \sigma \vdash s : \tau}{\Delta \mid \Gamma \vdash \mathbf{let}\, x = t\, \mathbf{in}\, s : \tau} \qquad \frac{(c \in \mathtt{R})}{\Delta \mid \Gamma \vdash \underline{c} : \mathbf{real}}$$

$$\frac{\{\Delta \mid \Gamma \vdash t_i : \mathbf{real}\}_{i=1}^{n} \quad (\mathrm{op} \in \mathrm{Op}_n)}{\Delta \mid \Gamma \vdash \mathrm{op}(t_1, \ldots, t_n) : \mathbf{real}} \qquad \frac{\Delta \mid \Gamma \vdash t : \mathbf{0}}{\Delta \mid \Gamma \vdash \mathbf{case}\, t\, \mathbf{of}\, \{\,\} : \tau}$$

$$\frac{\Delta \mid \Gamma \vdash t : \tau}{\Delta \mid \Gamma \vdash \mathbf{inl}\, t : \tau \sqcup \sigma} \qquad \frac{\Delta \mid \Gamma \vdash t : \tau}{\Delta \mid \Gamma \vdash \mathbf{inr}\, t : \tau \sqcup \sigma}$$

$$\frac{\Delta \mid \Gamma \vdash r : \sigma \sqcup \rho \quad \Delta \mid \Gamma, x : \sigma \vdash t : \tau \quad \Delta \mid \Gamma, y : \rho \vdash s : \tau}{\Delta \mid \Gamma \vdash \mathbf{case}\, r\, \mathbf{of}\, \{\mathbf{inl}\, x \to t \mid \mathbf{inr}\, y \to s\} : \tau} \qquad \frac{}{\Delta \mid \Gamma \vdash \langle\,\rangle : \mathbf{1}}$$

$$\frac{\Delta \mid \Gamma \vdash t : \tau \quad \Delta \mid \Gamma \vdash s : \sigma}{\Delta \mid \Gamma \vdash \langle t, s\rangle : \tau \times \sigma} \qquad \frac{\Delta \mid \Gamma \vdash r : \sigma \times \rho \quad \Delta \mid \Gamma, x : \sigma, y : \rho \vdash t : \tau}{\Delta \mid \Gamma \vdash \mathbf{case}\, r\, \mathbf{of}\, \langle x, y\rangle \to t : \tau}$$

$$\frac{\Delta \mid \Gamma, x : \sigma \vdash t : \tau}{\Delta \mid \Gamma \vdash \lambda x.t : \sigma \to \tau} \qquad \frac{\Delta \mid \Gamma \vdash t : \sigma \to \tau \quad \Delta \mid \Gamma \vdash s : \sigma}{\Delta \mid \Gamma \vdash t\, s : \tau} \qquad \frac{\Delta \mid \Gamma \vdash r : \mathbf{real}}{\Delta \mid \Gamma \vdash \mathbf{sign}\, r : \mathbf{1} \sqcup \mathbf{1}}$$

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

FIGURE B.1. Typing rules for the source language, where $\mathtt{R} \subseteq \mathbb{R}$ is a fixed set of real numbers containing 0.

$$\frac{(i \in \mathbb{N}^*)}{\Delta \mid \Gamma \vdash \overline{e}_i : \mathbf{vect}} \qquad \frac{}{\Delta \mid \Gamma \vdash \overline{0} : \mathbf{vect}} \qquad \frac{\Delta \mid \Gamma \vdash t : \mathbf{vect} \quad \Delta \mid \Gamma \vdash s : \mathbf{vect}}{\Delta \mid \Gamma \vdash t + s : \mathbf{vect}}$$

$$\frac{\Delta \mid \Gamma \vdash t : \mathbf{vect} \quad \Delta \mid \Gamma \vdash s : \mathbf{real}}{\Delta \mid \Gamma \vdash t * s : \mathbf{vect}} \qquad \frac{(i \in \mathbb{N}^*) \quad \Delta \mid \Gamma \vdash t : \mathbf{vect}}{\Delta \mid \Gamma \vdash \mathfrak{h}_i t : \mathbf{real}^i}$$

FIGURE B.2. Extra typing rules for the target language, where we denote $\mathbb{N}^* := \mathbb{N} - \{0\}$, $\mathbf{real}^1 := \mathbf{real}$ and $\mathbf{real}^{i+1} = \mathbf{real}^i \times \mathbf{real}$.

Appendix A. Source language

$$\frac{((x:\tau)\in\Gamma)}{\Delta \mid \Gamma \vdash x:\tau} \qquad \frac{\Delta \mid \Gamma \vdash t:\sigma \quad \Delta \mid \Gamma, x:\sigma \vdash s:\tau}{\Delta \mid \Gamma \vdash \mathbf{let}\, x = t\, \mathbf{in}\, s:\tau} \qquad \frac{(c\in \mathtt{R})}{\Delta \mid \Gamma \vdash \underline{c}:\mathbf{real}}$$

$$\frac{\{\Delta \mid \Gamma \vdash t_i:\mathbf{real}\}_{i=1}^{n} \quad (\mathrm{op}\in \mathrm{Op}_n)}{\Delta \mid \Gamma \vdash \mathrm{op}(t_1,\dots,t_n):\mathbf{real}} \qquad \frac{\Delta \mid \Gamma \vdash t:\mathbf{0}}{\Delta \mid \Gamma \vdash \mathbf{case}\, t\, \mathbf{of}\, \{\,\}:\tau}$$

$$\frac{\Delta \mid \Gamma \vdash t:\tau}{\Delta \mid \Gamma \vdash \mathbf{inl}\, t:\tau\sqcup\sigma} \qquad \frac{\Delta \mid \Gamma \vdash t:\tau}{\Delta \mid \Gamma \vdash \mathbf{inr}\, t:\tau\sqcup\sigma}$$

$$\frac{\Delta \mid \Gamma \vdash r:\sigma\sqcup\rho \quad \Delta \mid \Gamma, x:\sigma \vdash t:\tau \quad \Delta \mid \Gamma, y:\rho \vdash s:\tau}{\Delta \mid \Gamma \vdash \mathbf{case}\, r\, \mathbf{of}\, \{\mathbf{inl}\, x \to t \mid \mathbf{inr}\, y \to s\}:\tau} \qquad \frac{}{\Delta \mid \Gamma \vdash \langle\,\rangle:\mathbf{1}}$$

$$\frac{\Delta \mid \Gamma \vdash t:\tau \quad \Delta \mid \Gamma \vdash s:\sigma}{\Delta \mid \Gamma \vdash \langle t,s\rangle:\tau\times\sigma} \qquad \frac{\Delta \mid \Gamma \vdash r:\sigma\times\rho \quad \Delta \mid \Gamma, x:\sigma, y:\rho \vdash t:\tau}{\Delta \mid \Gamma \vdash \mathbf{case}\, r\, \mathbf{of}\, \langle x,y\rangle \to t:\tau}$$

$$\frac{\Delta \mid \Gamma, x:\sigma \vdash t:\tau}{\Delta \mid \Gamma \vdash \lambda x.t:\sigma\to\tau} \qquad \frac{\Delta \mid \Gamma \vdash t:\sigma\to\tau \quad \Delta \mid \Gamma \vdash s:\sigma}{\Delta \mid \Gamma \vdash t\, s:\tau} \qquad \frac{\Delta \mid \Gamma \vdash r:\mathbf{real}}{\Delta \mid \Gamma \vdash \mathbf{sign}\, r:\mathbf{1}\sqcup\mathbf{1}}$$

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

FIGURE A.1. Typing rules for the source language, where $\mathtt{R}\subseteq\mathbb{R}$ is a fixed set of real numbers containing 0. Types can use type variables α,β from the kinding context Δ.

$$\begin{aligned}
&\mathbf{let}\, x = v\, \mathbf{in}\, t = t[{}^{v}/_{x}]\\
&\mathbf{let}\, y = (\mathbf{let}\, x = t\, \mathbf{in}\, s)\, \mathbf{in}\, r = \mathbf{let}\, x = t\, \mathbf{in}\, (\mathbf{let}\, y = s\, \mathbf{in}\, r)\\
&\mathbf{case}\, \mathbf{inl}\, v\, \mathbf{of}\, \{\mathbf{inl}\, x \to t \mid \mathbf{inr}\, y \to s\} = t[{}^{v}/_{x}] \qquad t[{}^{v}/_{z}] \overset{\#x,y}{=} \mathbf{case}\, v\, \mathbf{of}\, \{\ \begin{matrix}\mathbf{inl}\, x \to t[{}^{\mathbf{inl}\, x}/_{z}]\\ \mid \mathbf{inr}\, y \to t[{}^{\mathbf{inr}\, y}/_{z}]\end{matrix}\ \}\\
&\mathbf{case}\, \mathbf{inr}\, v\, \mathbf{of}\, \{\mathbf{inl}\, x \to t \mid \mathbf{inr}\, y \to s\} = s[{}^{v}/_{y}]\\
&\mathbf{case}\, \langle v,w\rangle\, \mathbf{of}\, \langle x,y\rangle \to t = t[{}^{v}/_{x},{}^{w}/_{y}] \qquad t[{}^{v}/_{z}] \overset{\#x,y}{=} \mathbf{case}\, v\, \mathbf{of}\, \langle x,y\rangle \to t[{}^{\langle x,y\rangle}/_{z}]\\
&(\lambda x.t)\ v = t[{}^{v}/_{x}] \qquad v \overset{\#x}{=} \lambda x.v\, x\\
&\qquad \mathbf{case}\, \mathbf{roll}\, v\, \mathbf{of}\, \mathbf{roll}\, x \to t = t[{}^{v}/_{x}] \qquad t[{}^{v}/_{z}] \overset{\#x}{=} \mathbf{case}\, v\, \mathbf{of}\, \mathbf{roll}\, x \to t[{}^{\mathbf{roll}\, x}/_{z}]
\end{aligned}$$

FIGURE A.2. The standard $\beta\eta$-equational theory for a *CBV* language with recursive types. We write $\overset{\#x_1,\dots,x_n}{=}$ to indicate that the variables $x_1,\dots,x_n$ are fresh in the left hand side. In the second rule, x may not be free in r. Equations hold on pairs of terms of the same type.

generality. We call a pair $(\mathcal{V},\mathcal{T})$, where $\mathcal{V}$ is bicartesian closed and $\mathcal{T}$ is a $\mathcal{V}$-enriched monad, a *CBV pair*. In this setting, we call $\mathcal{V}$ the *category of values* and the corresponding Kleisli $\mathcal{V}$-category $\mathcal{C}$ the *category of computations*.

A *CBV pair morphism* between *CBV* pairs $(\mathcal{V},\mathcal{T})$ and $(\mathcal{V}',\mathcal{T})$ consists of a strictly bicartesian closed functor $H:\mathcal{V}\to\mathcal{V}'$ such that $H\eta = \eta'_H$ and $H\mathrm{m} = \mathrm{m}'_H$, where η,η' and m,m' are the respective units and multiplications of $\mathcal{T},\mathcal{T}'$.

2.1. Parametric types and type recursion. Let $(\mathcal{V},\mathcal{T})$ be a *CBV* pair. For each $n\in\mathbb{N}$, we can model $(n+1)$-variable $(\mathcal{V},\mathcal{T})$-parametric types as pairs $(E_{\mathcal{V}},E_{\mathcal{C}})$ of $\mathcal{V}$-enriched functors such that (2.1) commutes, where J is corresponding universal Kleisli $\mathcal{V}$-functor from $\mathcal{V}$ into the Kleisli $\mathcal{V}$-category $\mathcal{C}$. The 0-variable parametric types (2.3) are identified with objects of $\mathcal{V}$.

$$\begin{array}{ccc}(\mathcal{C}^{\mathrm{op}}\times\mathcal{C})^{n+1} & \xrightarrow{E_{\mathcal{C}}} & \mathcal{C}\\ {\scriptstyle (J^{\mathrm{op}}\times J)^{n+1}}\uparrow & & \uparrow{\scriptstyle J}\\ (\mathcal{V}^{\mathrm{op}}\times\mathcal{V})^{n+1} & \xrightarrow[E_{\mathcal{V}}]{} & \mathcal{V}\end{array} \qquad (2.1) \qquad \begin{array}{ccc}(\mathcal{C}^{\mathrm{op}}\times\mathcal{C})^{n} & \xrightarrow{\nu E_{\mathcal{C}}} & \mathcal{C}\\ {\scriptstyle (J^{\mathrm{op}}\times J)^{n}}\uparrow & & \uparrow{\scriptstyle J}\\ (\mathcal{V}^{\mathrm{op}}\times\mathcal{V})^{n} & \xrightarrow[\nu E_{\mathcal{V}}]{} & \mathcal{V}\end{array} \qquad (2.2)$$

We model *type recursion* in this setting. *Let* $\mathsf{Param}\,(\mathcal{V},\mathcal{T})$ *be the collection of all* $(\mathcal{V},\mathcal{T})$*-parametric types.* A *free type recursion* for $(\mathcal{V},\mathcal{T})$ is a pair $\underline{\nu} = (\nu,\underline{\mathsf{roll}})$, where: **(1)** ν is an operator that is the identity on 0-variable parametric types and associates each $(n+1)$-variable parametric type (2.1) with an n-variable parametric type (2.2); **(2)** $\underline{\mathsf{roll}}$ is a collection (2.4) of natural transformations such that (2.5) is invertible: namely, $J\left(\mathsf{roll}^{E}\right)$ is a natural isomorphism.

$$\left((\mathcal{V}^{\mathrm{op}}\times\mathcal{V})^{0}\to\mathcal{V},(\mathcal{C}^{\mathrm{op}}\times\mathcal{C})^{0}\to\mathcal{C}\right) \qquad (2.3)$$

$$\underline{\mathsf{roll}} = \left(\mathsf{roll}^{E}\right)_{E=(E_{\mathcal{V}},E_{\mathcal{C}})\in\mathsf{Param}(\mathcal{V},\mathcal{T})} \qquad (2.4)$$

$$\begin{array}{ccc}(\mathcal{V}^{\mathrm{op}}\times\mathcal{V})^{n} & \xrightarrow{\left(\mathrm{id},\nu E_{\mathcal{V}}^{\mathrm{op}},\nu E_{\mathcal{V}}\right)} & (\mathcal{V}^{\mathrm{op}}\times\mathcal{V})^{n+1}\\ & {\scriptstyle \nu E_{\mathcal{V}}}\searrow \quad \Leftarrow^{\mathsf{roll}^{E}} & \downarrow{\scriptstyle E_{\mathcal{V}}}\\ \mathcal{C} & \xleftarrow{\qquad} & \mathcal{V}\end{array} \qquad (2.5)$$

A *CBV* pair endowed with a free type recursion is our basic definition of model for our language with tuple, variant, function and recursive types.

Definition 2.1 ($rCBV$ model morphism)**.** An *rCBV model* is a triple $(\mathcal{V},\mathcal{T},\underline{\nu})$ where $(\mathcal{V},\mathcal{T})$ is a *CBV* pair and $\underline{\nu}$ is a free type recursion for $(\mathcal{V},\mathcal{T})$.

An *rCBV model morphism* between *rCBV* models $(\mathcal{V},\mathcal{T},\underline{\nu})$ and $(\mathcal{V}',\mathcal{T}',\underline{\nu}')$ consists of a *CBV* pair morphism $H:\mathcal{V}\to\mathcal{V}'$ such that, for each $n\in\mathbb{N}$ and each pair (E,E') of $(n+1)$-variable parametric types satisfying $E'_{\mathcal{V}'}\circ(H^{\mathrm{op}}\times H)^{n+1} = H\circ E_{\mathcal{V}}$, we have that $\nu E'_{\mathcal{V}'}\circ(H^{\mathrm{op}}\times H)^{n} = H\circ\nu E_{\mathcal{V}}$ and

$H\left(\mathsf{roll}^E\right) = \mathsf{roll}^E_{(H^{\mathrm{op}}\times H)^n}$. *We denote by* $\mathfrak{C}_{\mathcal{RBV}}$ *the category of rCBV models and rCBV model morphisms.*

2.2. Concrete models based on ω-cpos. Herein, for simplicity's sake, we avoid the generality of *bilimit compact expansions* by considering a subclass of concrete models, the *rCBV* $\boldsymbol{\omega}$**Cpo**-pairs.*

Let us write $\boldsymbol{\omega}$**Cpo** for the usual category of ω-complete partial orders and monotone ω-continuous functions. Recall that it is a complete, cocomplete cartesian closed category. An $\boldsymbol{\omega}$**Cpo**-category $\mathcal{V}$ is $\boldsymbol{\omega}$**Cpo**-*cartesian closed* if $\mathcal{V}$ has finite $\boldsymbol{\omega}$**Cpo**-products and, moreover, for each object $B \in \mathcal{V}$, the $\boldsymbol{\omega}$**Cpo**-functor $(B \times -)$ has a right $\boldsymbol{\omega}$**Cpo**-adjoint $\mathcal{V}\,[B, -]$.

A morphism j *in an* $\boldsymbol{\omega}$**Cpo**-*category* $\mathcal{B}$ *is full* if $\mathcal{C}\,(B, j)$ is a full morphism in $\boldsymbol{\omega}$**Cpo** for any $B \in \mathcal{C}$. Moreover, an *embedding-projection-pair (ep-pair)* $u : A \overset{\hookrightarrow}{\leftarrow} B$ in an $\boldsymbol{\omega}$**Cpo**-category $\mathcal{C}$ is a pair $u = (u^e, u^p)$ consisting of a $\mathcal{C}$-morphism $u^e : A \to B$, the *embedding*, and a $\mathcal{C}$-morphism $u^p : B \to A$, the *projection*, such that $u^e \circ u^p \leq \mathrm{id}$ and $u^p \circ u^e = \mathrm{id}$. A zero object† $\mathfrak{O}$ in an $\boldsymbol{\omega}$**Cpo**-category $\mathcal{C}$ is an *ep-zero object* if, for any object A, the pair $\iota_A = (\iota^e : \mathfrak{O} \to A, \iota^p : A \to \mathfrak{O})$ consisting of the unique morphisms is an ep-pair.

Definition 2.2 ($rCBV$ $\boldsymbol{\omega}$**Cpo**-pair)**.** An $rCBV$ $\boldsymbol{\omega}$**Cpo**-pair is a CBV pair $(\mathcal{V}, \mathcal{T})$ such that, denoting by $J : \mathcal{V} \to \mathcal{C}$ the corresponding universal Kleisli $\mathcal{V}$-functor,

- r1. $\mathcal{V}$ is a cocomplete $\boldsymbol{\omega}$**Cpo**-cartesian closed category;‡
- r2. the unit of $\mathcal{T}$ is pointwise a full morphism (hence, J is a locally full $\boldsymbol{\omega}$**Cpo**-functor);
- r3. $\mathcal{C}$ has an ep-zero object $\mathfrak{O} = J\,(\mathsf{0})$, where $\mathsf{0}$ is initial in $\mathcal{V}$;
- r4. whenever $u : J(A) \overset{\hookrightarrow}{\leftarrow} J(B)$ is an ep-pair in $\mathcal{C}$, there is one morphism $\hat{u} : A \to B$ in $\mathcal{V}$ such that $J\,(\hat{u}) = u^e$.

An $rCBV$ $\boldsymbol{\omega}$**Cpo**-*pair morphism* from $(\mathcal{V}, \mathcal{T})$ into $(\mathcal{V}', \mathcal{T}')$ is an $\boldsymbol{\omega}$**Cpo**-functor $H : \mathcal{V} \to \mathcal{V}'$ that strictly preserves $\boldsymbol{\omega}$**Cpo**-colimits, and whose underlying functor is a CBV pair morphism. This defines a category of $rCBV$ $\boldsymbol{\omega}$**Cpo**-pairs, denoted herein by $\boldsymbol{\omega}$**CPO**-$\mathfrak{C}_{r\mathcal{BV}}$.

*See [17, 4.2.2] or [33, Sect. 8] for the general setting of bilimit compact expansions.

†Recall that a *zero object* is an object that is both initial and terminal.

‡$\mathcal{V}$ is, hence, $\boldsymbol{\omega}$**Cpo**-cocomplete as well.

[19] F. Lucatelli Nunes and M. Vákár. CHAD for Expressive Total Languages. *arXiv e-prints*, page arXiv:2110.00446, October 2021.

[20] F. Lucatelli Nunes and M. Vákár. Automatic Differentiation for ML-family languages: correctness via logical relations. *arXiv e-prints*, pages –, October 2022.

[21] D. Mazza and M. Pagani. Automatic differentiation in PCF. *Proc. ACM Program. Lang.*, 5(POPL):1–27, 2021.

[22] E. Meijer. Behind every great deep learning framework is an even greater programming languages concept (keynote). In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1–1, 2018.

[23] E. Moggi. Computational lambda-calculus and monads. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science (LICS '89), Pacific Grove, California, USA, June 5-8, 1989*, pages 14–23. IEEE Computer Society, 1989.

[24] B. Pearlmutter and J. Siskind. Reverse-mode AD in a functional framework: Lambda the ultimate backpropagator. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 30(2):7, 2008.

[25] A. Pitts. Relational properties of domains. *Inform. Comput.*, 127(2):66–90, 1996.

[26] G. Plotkin. Some principles of differential programming languages. *Invited talk, POPL*, 2018, 2018.

[27] A. Shaikhha, A. Fitzgibbon, D. Vytiniotis, and S. Peyton Jones. Efficient differentiable programming in a functional array-processing language. *Proceedings of the ACM on Programming Languages*, 3(ICFP):97, 2019.

[28] S. Shalev-Shwartz et al. Online learning and online convex optimization. *Foundations and Trends® in Machine Learning*, 4(2):107–194, 2012.

[29] T. Smeding and M. Vákár. Efficient Dual-Numbers Reverse AD via Well-Known Program Transformations. *arXiv e-prints*, page arXiv:2207.03418, July 2022.

[30] R. Socher, C. C. Lin, C. Manning, and A. Y. Ng. Parsing natural scenes and natural language with recursive neural networks. In *Proceedings of the 28th international conference on machine learning (ICML-11)*, pages 129–136, 2011.

[31] K. S. Tai, R. Socher, and C. D. Manning. Improved semantic representations from tree-structured long short-term memory networks. *arXiv preprint arXiv:1503.00075*, 2015.

[32] P. Tsiros, F. Y. Bois, A. Dokoumetzidis, G. Tsiliki, and H. Sarimveis. Population pharmacokinetic reanalysis of a diazepam pbpk model: a comparison of stan and gnu mcsim. *Journal of Pharmacokinetics and Pharmacodynamics*, 46(2):173–192, 2019.

[33] M. Vákár. Denotational correctness of forward-mode automatic differentiation for iteration and recursion. *arXiv preprint arXiv:2007.05282*, 2020.

[34] M. Vákár. Reverse AD at higher types: pure, principled and denotationally correct. In *Programming languages and systems. 30th European symposium on programming, ESOP 2021, held as part of the European joint conferences on theory and practice of software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 – April 1, 2021. Proceedings*, pages 607–634. Cham: Springer, 2021.

[35] M. Vákár and T. Smeding. CHAD: combinatory homomorphic automatic differentiation. *ACM Trans. Program. Lang. Syst.*, 44(3):20:1–20:49, 2022.

[36] X. Zhang, L. Lu, and M. Lapata. Top-down tree long short-term memory networks. In *Proceedings of NAACL-HLT*, pages 310–320, 2016.

References

[1] M. Abadi and G. Plotkin. A simple differentiable programming language. In *Proc. POPL 2020*. ACM, 2020.

[2] A. J. Ahmed. Step-indexed syntactic logical relations for recursive and quantified types. In P. Sestoft, editor, *Programming Languages and Systems, 15th European Symposium on Programming, ESOP 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 27-28, 2006, Proceedings*, volume 3924 of *Lecture Notes in Computer Science*, pages 69–83. Springer, 2006.

[3] G. Barthe, R. Crubillé, U. D. Lago, and F. Gavazzo. On the versatility of open logical relations: Continuity, automatic differentiation, and a containment theorem. In *Proc. ESOP 2020*. Springer, 2020. To appear.

[4] T. Beck and H. Fischer. The if-problem in automatic differentiation. *Journal of Computational and Applied Mathematics*, 50(1-3):119–131, 1994.

[5] M. Betancourt. Double-pareto lognormal distribution in stan, Aug 2019.

[6] A. Brunel, D. Mazza, and M. Pagani. Backpropagation in the simply typed lambda-calculus with linear negation. In *Proc. POPL 2020*, 2020.

[7] B. Carpenter, M. Hoffman, M. Brubaker, D. Lee, P. Li, and M. Betancourt. The Stan math library: Reverse-mode automatic differentiation in C++. *arXiv preprint arXiv:1509.07164*, 2015.

[8] E. Dubuc. *Kan extensions in enriched category theory*. Lecture Notes in Mathematics, Vol. 145. Springer-Verlag, Berlin-New York, 1970.

[9] C. Elliott. The simple essence of automatic differentiation. *Proceedings of the ACM on Programming Languages*, 2(ICFP):70, 2018.

[10] M. Fiore and G. Plotkin. An axiomatisation of computationally adequate domain theoretic models of fpc. In *Proceedings Ninth Annual IEEE Symposium on Logic in Computer Science*, pages 92–102. IEEE, 1994.

[11] S. Flaxman, S. Mishra, A. Gandy, H. J. T. Unwin, T. A. Mellan, H. Coupland, C. Whittaker, H. Zhu, T. Berah, J. W. Eaton, et al. Estimating the effects of non-pharmaceutical interventions on covid-19 in europe. *Nature*, 584(7820):257–261, 2020.

[12] B. Goodrich. Conway-maxwell-poisson distribution, Oct 2017.

[13] M. Huot, S. Staton, and M. Vákár. Correctness of automatic differentiation via diffeologies and categorical gluing. In J. Goubault-Larrecq and B. König, editors, *Foundations of Software Science and Computation Structures - 23rd International Conference, FOSSACS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings*, volume 12077 of *Lecture Notes in Computer Science*, pages 319–338. Springer, 2020.

[14] M. Huot, S. Staton, and M. Vákár. Higher order automatic differentiation of higher order functions. *CoRR*, abs/2101.06757, 2021.

[15] E. Jeong, J. Jeong, S. Kim, G.-I. Yu, and B.-G. Chun. Improving the expressiveness of deep learning frameworks with recursion. In *Proceedings of the Thirteenth EuroSys Conference*, pages 1–13, 2018.

[16] F. Krawiec, S. P. Jones, N. Krishnaswami, T. Ellis, R. A. Eisenberg, and A. W. Fitzgibbon. Provably correct, asymptotically efficient, higher-order reverse-mode automatic differentiation. *Proc. ACM Program. Lang.*, 6(POPL):1–30, 2022.

[17] P. Levy. *Call-by-push-value: A Functional/imperative Synthesis*, volume 2. Springer Science & Business Media, 2012.

[18] P. Levy, J. Power, and H. Thielecke. Modelling environments in call-by-value programming languages. *Information and computation*, 185(2):182–210, 2003.

Every $rCBV$ $\boldsymbol{\omega}$**Cpo**-pair $(\mathcal{V}, \mathcal{T})$ has an underlying $rCBV$ model. Namely, we have a canonical free type recursion $\underline{\nu}_\omega = \left(\nu_\omega, \omega\mathsf{roll}_A^E\right)$ where ν_ω is constructed out of (bi)limits of chains of ep-pairs (see [20, Section 8], the extended version of the paper). This construction extends to a functor $\mathcal{U}_{r\mathcal{BV}} : \boldsymbol{\omega}\mathbf{CPO}\text{-}\mathfrak{C}_{r\mathcal{BV}} \to \mathfrak{C}_{\mathcal{RBV}}$, which shows how our concrete models are indeed $rCBV$ models.

2.3. Syntax as freely generated $rCBV$ models. In Section 4, we will consider the syntax of an ML-family programming language with recursive types. It is generated from certain *primitive types* (like a type **real** for real numbers) and certain *primitive operations* op (e.g. mathematical operations like $\sin, \cos, \exp, (+), (*)$, etc.). We can consider the syntax of our languages as $rCBV$ models by taking the types τ of the language as objects and equational equivalence classes of programs $x : \tau \vdash t : \sigma$ as morphisms $\tau \to \sigma$. This is a freely generated $rCBV$ model in the sense that we get a unique $rCBV$ model morphism to any $rCBV$ model once we fix the image of all primitive types and operations in a consistent way. We call these *freely generated $rCBV$ models on a language syntactic $rCBV$ models.*

3. Subscone

We establish, here, the basic categorical framework underlying the logical relations *(LR)* argument. Our general view is that the categorical approach to semantic logical relations relies on studying principled ways to construct concrete categorical semantics out of elementary ones. This construction should be informed of the desired properties to be proved: so that the resulting semantics assures us of the property we want to establish in each setting.

The first step is to choose a basic concrete categorical semantics for our language. For CBV languages with recursive types like ours, a *basic* concrete model usually consists of an $rCBV$ $\boldsymbol{\omega}$**Cpo**-pair. We are particularly interested in the elementary $rCBV$ $\boldsymbol{\omega}$**Cpo**-pair $(\boldsymbol{\omega}\mathbf{Cpo}^n, (-)_\perp) = (\boldsymbol{\omega}\mathbf{Cpo}, (-)_\perp)^n$ for some $n \in \mathbb{N}$, where $(-)_\perp$ is the usual partiality $\boldsymbol{\omega}$**Cpo**-monad that freely adds a least element $\perp$ to each ω-cpo.

It is tempting to add or consider more structured semantics and appeal to more general theories. However, we believe that adding structure beforehand is an *ad hoc* anticipation of constructing our semantic logical relations proof, which is avoidable if we have powerful enough principled techniques.

Given a chosen basic $rCBV$ $\boldsymbol{\omega}\mathbf{Cpo}$-pair $(\mathcal{V}, \mathcal{T})$, with a well defined semantics $[\![-]\!]$ for our language, the second step is to establish the base LR $\boldsymbol{\omega}\mathbf{Cpo}$-functors: these are suitable $\boldsymbol{\omega}\mathbf{Cpo}$-functors $G : \mathcal{V} \to \boldsymbol{\omega}\mathbf{Cpo}$ such that we can express the property we want to prove starting from predicates over $G(Z) \in \boldsymbol{\omega}\mathbf{Cpo}$.

In this direction, in the context of semantic open logical relations (and in our AD setting), we often want to consider the family of $\boldsymbol{\omega}\mathbf{Cpo}$-functors given by $(\mathcal{V}([\![\tau]\!], -) : \mathcal{V} \to \boldsymbol{\omega}\mathbf{Cpo})_{\tau \in \mathsf{T}_p}$ indexed by a subset of types τ of the language.

Given such a family of $\boldsymbol{\omega}\mathbf{Cpo}$-functors $(G_\tau : \mathcal{V} \to \boldsymbol{\omega}\mathbf{Cpo})_{\tau \in \mathsf{T}_p}$, we consider the $\boldsymbol{\omega}\mathbf{Cpo}$-scone along G_τ (for each τ): these are the comma $\boldsymbol{\omega}\mathbf{Cpo}$-categories $\boldsymbol{\omega}\mathbf{Cpo} \downarrow G_\tau$ whose definition we recall below.

- The objects of $\boldsymbol{\omega}\mathbf{Cpo} \downarrow G_\tau$ are triples $(D \in \boldsymbol{\omega}\mathbf{Cpo}, C \in \mathcal{V}, j : D \to G(C))$ in which j is a morphism of $\boldsymbol{\omega}\mathbf{Cpo}$;
- a morphism $(D, C, j) \to (D', C', h)$ between objects of $\boldsymbol{\omega}\mathbf{Cpo} \downarrow G_\tau$ is a pair (3.1) making (3.3) commutative in $\mathcal{D}$;
- if (3.1) and (3.2) are two morphisms in $\boldsymbol{\omega}\mathbf{Cpo} \downarrow G_\tau ((D, C, j), (D', C', h))$, we have that $\alpha \leq \beta$ if $\alpha_0 \leq \beta_0$ in $\boldsymbol{\omega}\mathbf{Cpo}$ and $\alpha_1 \leq \beta_1$ in $\mathcal{V}$.

$$\alpha = (\alpha_0 : D \to D', \alpha_1 : C \to C') \quad (3.1) \qquad \beta = (\beta_0 : D \to D', \beta_1 : C \to C') \quad (3.2)$$

$$\begin{array}{ccc} D & \xrightarrow{\alpha_0} & D' \\ {\scriptstyle j}\downarrow & & \downarrow{\scriptstyle h} \\ G_\tau(C) & \xrightarrow[G_\tau(\alpha_1)]{} & G_\tau(C') \end{array} \quad (3.3)$$

Provided that G_τ is a right $\boldsymbol{\omega}\mathbf{Cpo}$-adjoint, the forgetful $\boldsymbol{\omega}\mathbf{Cpo}$-functor $\mathcal{L}_\tau : \boldsymbol{\omega}\mathbf{Cpo} \downarrow G_\tau \to \boldsymbol{\omega}\mathbf{Cpo} \times \mathcal{V}$ is $\boldsymbol{\omega}\mathbf{Cpo}$-comonadic and $\boldsymbol{\omega}\mathbf{Cpo}$-monadic. We can conclude, then, that it creates (and strictly preserves) $\boldsymbol{\omega}\mathbf{Cpo}$-colimits and limits. Moreover, $\boldsymbol{\omega}\mathbf{Cpo} \downarrow G_\tau$ is $\boldsymbol{\omega}\mathbf{Cpo}$-bicartesian closed. This is the $\boldsymbol{\omega}\mathbf{Cpo}$-enriched version of the results presented in [19, Section 9] (see [20, Appx. C]).

In order to proceed with a proof-irrelevant approach, we consider the subscone: namely, $\mathbf{Sub}(\mathcal{D} \downarrow G_\tau)$ is, herein, by definition the full $\boldsymbol{\omega}\mathbf{Cpo}$-subcategory of $\boldsymbol{\omega}\mathbf{Cpo} \downarrow G_\tau$ whose objects are triples $(D \in \mathcal{V}, C \in \boldsymbol{\omega}\mathbf{Cpo}, j)$ where j is a full morphism. $\mathbf{Sub}(\mathcal{D} \downarrow G_\tau)$ is, then, a full reflective and replete $\boldsymbol{\omega}\mathbf{Cpo}$-subcategory of $\boldsymbol{\omega}\mathbf{Cpo} \downarrow G_\tau$. Moreover, we have that:

Theorem 3.1. $\mathbf{Sub}(\boldsymbol{\omega}\mathbf{Cpo} \downarrow G_\tau)$ *is cocomplete and* $\boldsymbol{\omega}\mathbf{Cpo}$*-cartesian closed. Moreover, the forgetful* $\boldsymbol{\omega}\mathbf{Cpo}$*-functor* $\underline{\mathcal{L}}_\tau : \mathbf{Sub}(\boldsymbol{\omega}\mathbf{Cpo} \downarrow G_\tau) \to \mathcal{V}$ *is strictly* $\boldsymbol{\omega}\mathbf{Cpo}$*-colimit preserving, cartesian closed and locally full (hence, faithful).*

The reader interested in further considerations on the result above may take a look at [20, Section 6]. Theorem 3.1 shows how $\mathbf{Sub}(\boldsymbol{\omega}\mathbf{Cpo} \downarrow G_\tau)$ already

usually can interpret CBV languages with recursive types in a simple enough $rCBV$ $\boldsymbol{\omega}\mathbf{Cpo}$-pairs. We believe that working with this special case of the semantics significantly simplifies our presentation. We leave the presentation of the results in the setting of bilimit expansions for future work (if we find a useful setting where our current approach does not apply).

The use of subscone instead of the scone was a matter of presentation as well. Everything we did could be done for the scone (the comma category). Although it takes more work to establish it, the subscone provides us with simpler verifications. It also gives the proof-irrelevant approach.

Finally, we hope that our work adds to the existing body of programming languages literature on automatic differentiation and recursion (and recursive types). In particular, we believe that it provides a simple, principled denotational explaination of how AD and expressive partial language features should interact. We plan to use it to generalise and prove correct the more advanced AD technique CHAD [34, 35, 19] when applied to languages with partial features.

10. Related Work

There has recently been a flurry of work studying AD from a programming language point of view, a lot of it focussing on functional formulations of AD and their correctness. Examples of such papers are [24, 9, 27, 6, 1, 13, 21, 34, 19, 14, 35, 16, 29]. Of these papers, [24, 1, 21, 29] are particularly relevant as they also consider automatic differentiation of languages with partial features. Here, [24] considers an implementation that differentiates recursive programs and the implementation of [29] even differentiates code that uses recursive types. They do not give correctness proofs, however.

The present paper can be seen as giving a correctness proof of the techniques implemented by [29]. [1] does give a denotational correctness proof of AD on a first-order functional language with (first-order) recursion. The first-orderness of the language allows the proof to proceed by plain induction rather than needing logical technique. [21] proves the correctness of basically the same AD algorithms that we consider in this paper when restricted to PCF with a base type of real numbers and a real conditional. Their proof relies on operational semantic techniques. Our contribution is to give an alternative denotational argument, which we believe is simple and systematic, and to extend it to apply to languages which, additionally, have the complex features of recursively defined datastructures that we find in realistic ML-family languages.

(where τ and σ are data types) in the source language, we get a program $x : \mathcal{D}(\tau) \vdash \mathcal{D}(t) : \mathcal{D}(\sigma)$ in the target language, which, by Theorem 7.3, satisfies the following properties: **(A)** $[\![t]\!] : \coprod_{r\in K} \mathbb{R}^{n_r} \to \left(\coprod_{j\in L} \mathbb{R}^{m_j}\right)_\perp$ is differentiable; **(B)** if $y \in \mathbb{R}^{n_i} \cap [\![t]\!]^{-1}(\mathbb{R}^{m_j}) = W_j$ for some $i \in K$ and $j \in L$, we have that, (8.1) holds, for any $w \in \mathbb{R}^{n_i}$, where $[\![t]\!]'(y) : \mathbb{R}^{n_i} \to \mathbb{R}^{m_j}$ is the derivative of $[\![t]\!]|_{W_j} : W_j \to \mathbb{R}^{m_j}$ at y.

$$[\![\mathcal{D}(t)]\!]_1 (\phi_{n_i,1}(y, w)) = \phi_{l,1}([\![t]\!](y), [\![t]\!]'(y)(w)) \tag{8.1}$$

8.2. Correctness of the dual numbers reverse AD ($k = \infty$). The following shows how our macro encompasses reverse mode AD. We assume that **vect** implements the vector space $\mathbb{R}^\infty$ (representing the case of a type of dynamically sized array of cotangents).

For each $s \in \mathbb{N} \cup \{\infty\}$, we consider the respective (co)projections $\mathfrak{p}_{\infty\to s}$, and we define the morphism $\mathbf{wrap}_s \stackrel{\text{def}}{=} (\pi_j, \overline{e}_j)_{j\in\mathbb{N}} : \mathbf{real}^s \to (\mathbf{real} \times \mathbf{vect})^s$ in $\mathbf{Syn}_V^{\mathbf{tr}}$. For a program $x : \mathbf{real}^s \vdash t : \mathbf{real}^l$ (where $s, l \in \mathbb{N}^*$), we have that, for any $y \in [\![t]\!]^{-1}(\mathbb{R}^l) \subseteq \mathbb{R}^s$, (8.2) holds by Theorem 7.3. This gives the transpose derivative $\mathfrak{p}_{s\to\infty}[\![t]\!]'(y)^t$ as something of the type $\mathbf{vect}^l$. The type can be fixed by using the handler $\mathfrak{h}_s$ (see [20]).

$$[\![\mathcal{D}(t) \circ \mathbf{wrap}_s]\!]_\infty (y) = \phi_{l,\infty}\left([\![t]\!](y), \mathfrak{p}_{s\to\infty}[\![t]\!]'(y)^t\right) \tag{8.2}$$

By Theorem 7.3, it is straightforward to generalize the correctness statements above to more general data types σ.

9. Final remarks

This work improved on the proof previously given in [33]: we gave a simple correctness proof of dual numbers forward and reverse AD for realistic ML-family languages by making use of nimble new logical relations techniques for recursive types and partial differentiable functions. In particular, we have simplified the argument to no longer depend on diffeological or sheaf-structure and to have it apply to arbitrary differentiable (rather than merely smooth) operations. We have further simplified the subsconing technique for recursive types.

Although we can formulate the subsconing technique presented here in more general settings, such as the setting of bilimit expansions [17], we opted for a simpler presentation – making use of the $rCBV$ $\boldsymbol{\omega}\mathbf{Cpo}$-pairs introduced herein. This approach is enough for semantic (open) logical relations, since we

yields a model for the category of values of our language. The remaining step consists in giving the conditions under which a $\mathbf{Sub}(\boldsymbol{\omega}\mathbf{Cpo} \downarrow G_\tau)$-monad $\mathcal{T}_\tau$ will give us an $rCBV$ $\boldsymbol{\omega}\mathbf{Cpo}$-pair and an $rCBV$ $\boldsymbol{\omega}\mathbf{Cpo}$-pair morphism from $(\mathbf{Sub}(\boldsymbol{\omega}\mathbf{Cpo} \downarrow G_\tau), \mathcal{T}_\tau)$ into $(\mathcal{V}, \mathcal{T})$.

3.1. Recursive types for lifted monads. In the context of Theorem 3.1, there might be canonical/universal liftings of the monad $\mathcal{T}$ along $\underline{\mathcal{L}}_\tau$. However, these are not necessarily the monads we want – since we are assuming that we started out from a very basic semantics, defined in an $rCBV$ $\boldsymbol{\omega}\mathbf{Cpo}$-pair $(\mathcal{V}, \mathcal{T})$. In this scenario, our lifting should be informed with the logical relations we want for the computations of primitive types.

Let $\mathcal{T}_\tau$ be a strong $\mathbf{Sub}(\boldsymbol{\omega}\mathbf{Cpo} \downarrow G_\tau)$-monad that is a lifting of $\mathcal{T}$ along $\underline{\mathcal{L}}_\tau$. This means that $\underline{\mathcal{L}}_\tau$ yields a CBV pair morphism (3.4). Assume that $\mathcal{T}_\tau$ satisfies the following two properties: **(A)** for each object (D, C, j) of $\mathbf{Sub}(\boldsymbol{\omega}\mathbf{Cpo} \downarrow G_\tau)$, the square induced by each component of the unit of $\mathcal{T}_\tau$ in $\boldsymbol{\omega}\mathbf{Cpo}$ is a pullback; and **(B)** $\mathcal{T}_\tau(\mathsf{0}, \mathsf{0}, j)$ is the terminal object in $\mathbf{Sub}(\boldsymbol{\omega}\mathbf{Cpo} \downarrow G_\tau)$. In this setting, we get that Theorem 3.2 holds (by the main result established in [20, 8.8]).

Theorem 3.2. $(\mathbf{Sub}(\boldsymbol{\omega}\mathbf{Cpo} \downarrow G_\tau), \mathcal{T}_\tau)$ *is an* $rCBV$ $\boldsymbol{\omega}\mathbf{Cpo}$*-pair. Moreover,* (3.4) *yields an* $rCBV$ $\boldsymbol{\omega}\mathbf{Cpo}$*-pair morphism.*

$$\underline{\mathcal{L}}_\tau : (\mathbf{Sub}(\boldsymbol{\omega}\mathbf{Cpo} \downarrow G_\tau), \mathcal{T}_\tau) \to (\mathcal{V}, \mathcal{T}) \tag{3.4}$$

In our use case of Theorem 3.2, the remaining step to define the logical relations is to define a morphism $[\![\mathrm{op}]\!]_\tau \in \mathbf{Sub}(\boldsymbol{\omega}\mathbf{Cpo} \downarrow G_\tau)$ for each primitive operation op in a compatible way. This yields an $rCBV$ model morphism $[\![-]\!]_\tau$ from the free $rCBV$ model on the syntax of our language to $\mathcal{U}_{r\mathcal{BV}}(\mathbf{Sub}(\boldsymbol{\omega}\mathbf{Cpo} \downarrow G_\tau), \mathcal{T}_\tau)$.

4. Syntax and AD macro for an ML-like language

We establish basic example of languages with type recursion where we can do automatic differentiation. Since it relates better to the efficient implementations we have in mind [29], we see *automatic differentiation* (AD) as a program transformation between two languages. The target language is a simple extension of the source language: we add an extra type **vect** of (co)tangent vectors.

4.1. Source language. For the source language, we consider a standard (coarse-grain) call-by-value language with ML-style polymorphism and type recursion in the sense of FPC [10]. The language is constructed over a ground type

real, certain real constants $\underline{c} \in \mathrm{Op}_0$, certain primitive operations $\mathrm{op} \in \mathrm{Op}_n$ for each nonzero natural number $n \in \mathbb{N}^*$, and **sign**. We denote $\mathrm{Op} := \bigcup_{n\in\mathbb{N}} \mathrm{Op}_n$.

real intends to implement floating point or exact real numbers. Moreover, for each $n \in \mathbb{N}$, the operations in Op_n intend to implement partially defined functions $\mathbb{R}^n \rightharpoonup \mathbb{R}$. Finally, **sign** intends to implement the partially defined function $\mathsf{sign} : \mathbb{R} \rightharpoonup \mathbb{R}$ defined in $\mathbb{R}^- \cup \mathbb{R}^+$ which takes $\mathbb{R}^-$ to -1 and $\mathbb{R}^+$ to 1.

We will later interpret these operations as morphisms $\mathbb{R}^n \to (\mathbb{R})_\perp$ in $\boldsymbol{\omega}\mathbf{Cpo}$ with domains of definition that are (topologically) open in $\mathbb{R}^n$, on which they are differentiable functions. These operations include, for example, unary operations on reals like $\exp, \log, \varsigma \in \mathrm{Op}_1$ (where we mean the mathematical sigmoid function $\varsigma(x) \stackrel{\text{def}}{=} \frac{1}{1+e^{-x}}$), binary operations on reals like $(+), (-), (*), (/) \in \mathrm{Op}_2$, or any other desired partially defined differentiable function $\mathbb{R}^n \to (\mathbb{R})_\perp$ (see Sect. 5).

We treat these operations in a schematic way as this reflects the reality of practical AD libraries, which are constantly being expanded with new primitive operations. The types τ, σ, ρ, values v, w, u, and computations t, s, r of our language are as follows - they are standard:

τ, σ, ρ	$::=$	types	$\mid \tau \to \sigma$	function	
	$\mid$ **real**	numbers	$\mid \alpha, \beta, \gamma$	type variables	
	$\mid$ **0** $\mid \tau \sqcup \sigma$	sums	$\mid \mu\alpha.\tau$	recursive type	
	$\mid$ **1** $\mid \tau_1 \times \tau_2$	products			
v, w, u	$::=$	values	$\mid \langle\rangle \mid \langle v, w\rangle$	tuples	
	$\mid x, y, z$	variables	$\mid \lambda x.t$	abstractions	
	$\mid \underline{c}$	constants	$\mid$ **roll** v	recursive intro	
	$\mid$ **inl** $v \mid$ **inr** v	inclusions			
t, s, r	$::=$	computations	$\mid$ **case** t **of** $\{\,\}$	sum match	
	$\mid x, y, z$	variables	$\mid \langle\rangle \mid \langle t, s\rangle$	tuples	
	$\mid$ **let** $t = x$ **in** s	sequencing	$\mid$ **case** s **of** $\langle x, y\rangle \to t$	product match	
	$\mid \underline{c}$	constant	$\mid \lambda x.t$	abstractions	
	$\mid \mathrm{op}(t_1, \ldots, t_n)$	operation	$\mid t\ s$	function app.	
	$\mid$ **inl** $t \mid$ **inr** t	inclusions	$\mid$ **sign** t	sign function	
	$\mid$ **case** r **of** $\{ \begin{smallmatrix} \textbf{inl}\, x \to t \\ \mid\ \textbf{inr}\, y \to s \end{smallmatrix} \}$	sum match	$\mid$ **roll** t	recursive intro	

We use the sugar $\textbf{if}\, r\, \textbf{then}\, t\, \textbf{else}\, s \stackrel{\text{def}}{=} \textbf{case}\, \textbf{sign}\, r\, \textbf{of}\, \{_ \to t \mid _ \to r\}$ for lazy real conditionals.

The typing rules for computations are standard, with the exception of the typing of constants and primitive operations which are listed in Fig. 4.1. (We

that (7.4) belongs to $\underline{\mathcal{P}_{s_i}\left(\coprod_{j\in L} \overline{[\![\mathbf{real}]\!]}_{s_i}^{l_j}\right)_\perp}$. By the observed above, this shows that $h \circ \iota_{\mathbb{R}^{s_i}}$ is differentiable and $\dot{h} \circ \iota_{(\mathbb{R}\times\mathbb{R}^k)^{s_i}}$ is its derivative. Since this results holds for every $i \in \mathfrak{L}$, the proof is complete. ∎

The commutativity of (7.2) implies that, for any morphism t of the syntax (the category $\mathbf{Syn}_V$), we have that the pair $([\![t]\!], [\![\mathbb{D}(t)]\!]_k)$ defines a morphism in $\overline{[\![t]\!]}_n$ in $\mathbf{Sub}(\boldsymbol{\omega}\mathbf{Cpo} \downarrow G_n)$ for every $n \in \mathbb{N}$. Therefore, by Theorem 7.1, we have:

Theorem 7.2. *Let $t : \tau \to \sigma$ be a morphism of $\mathbf{Syn}_V$, i.e. a program in our source language. If there are families $(s_r)_{r\in\mathfrak{L}}$ and $(l_j)_{j\in L}$ such that (7.5) and (7.6) hold for any $n \in \mathbb{N}$ (where $\cong$ is just an isomorphism induced by coprojections and projections), then $[\![t]\!]$ is differentiable and $[\![\mathbb{D}(t)]\!]_k = \mathfrak{d}^k([\![t]\!])$.*

$$\overline{[\![\tau]\!]}_n \cong \coprod_{r\in\mathfrak{L}} \overline{[\![\mathbf{real}]\!]}_n^{s_r} \qquad (7.5) \qquad\qquad \overline{[\![\sigma]\!]}_n \cong \coprod_{j\in L} \overline{[\![\mathbf{real}]\!]}_n^{l_j} \qquad (7.6)$$

Since $\overline{[\![-]\!]}_n$ is an *rCBV* model morphism, the hypothesis of Theorem 7.2 holds for any data types that do not involve function types (including types built using recursion) by Theorem C.3 of Appendix C. Therefore:

Theorem 7.3. *Assume that* **vect** *implements the vector space $\mathbb{R}^k$, for some $k \in \mathbb{N} \cup \{\infty\}$. For any program $x : \tau \vdash t : \sigma$ where τ, σ are data types not involving function types in their construction, we have that $[\![t]\!]$ is differentiable and, moreover, $[\![\mathcal{D}(t)]\!]_k = \mathfrak{d}^k([\![t]\!])$ provided that $\mathcal{D}$ is sound for primitives.*

8. Forward vs reverse mode AD – choosing k

We have so far been vague about how to choose k and whether we are considering forward or reverse AD. It turns out that our abstract development is enough for both AD methods, by choosing $k = 1$ for forward (no need for vectorised AD) and $k = \infty$ (AD with dynamically sized vectorized tangent types) for reverse AD. (Here, we remember that a practical implementation of dual numbers reverse AD like that of [29] would make use of a distributive law as a runtime optimisation to reach the correct computational complexity.)

8.1. Correctness of the dual numbers forward AD ($k = 1$). We assume that **vect** implements the vector space $\mathbb{R}$. It is straightforward to see that we get forward mode AD out of our macro $\mathcal{D}$: namely, for a program $x : \tau \vdash t : \sigma$

$\overline{[\![\mathbf{op}]\!]}_n : \overline{[\![\mathbf{real}]\!]}_n^m \to \mathcal{P}_n\left(\overline{[\![\mathbf{real}]\!]}_n\right)_\perp$ in $\mathbf{Sub}\left(\boldsymbol{\omega}\mathbf{Cpo} \downarrow G_n\right)$ by the chain-rule for derivatives. Analogously, we define compatible morphisms $\overline{[\![\mathbf{sign}]\!]}_n$ and $\overline{[\![\underline{c}]\!]}_n$ by the morphisms in $\mathbf{Sub}\left(\boldsymbol{\omega}\mathbf{Cpo} \downarrow G_n\right)$ defined resp. by the pairs $\left(\mathsf{sign}, \mathfrak{d}^k\left(\mathsf{sign}\right)\right)$ and $\left(\mathsf{c}, \mathfrak{d}^k\left(\mathsf{c}\right)\right)$.

$$\overline{[\![\mathbf{real}]\!]}_n \stackrel{\text{def}}{=} \left(\left\{\left(f : \mathbb{R}^n \to \mathbb{R}, f^*\right) : f \text{ is differentiable, } f^* = \mathfrak{D}^k f\right\}, \left(\mathbb{R}, \mathbb{R} \times \mathbb{R}^k\right), \text{incl.}\right) \tag{7.1}$$

By the universal property of the syntactic $rCBV$ model $\left(\mathbf{Syn}_V, \mathbf{Syn}_{\mathcal{S}}, \underline{\nu}_{\mathbf{Syn}}\right)$, there is only one $rCBV$ model morphism $\overline{[\![-]\!]}_n$ compatible with the assignment above and, moreover, we can conclude that (7.2) commutes.

$$\begin{array}{ccc} \left(\mathbf{Syn}_V, \mathbf{Syn}_{\mathcal{S}}, \underline{\nu}_{\mathbf{Syn}}\right) & \xrightarrow{(\mathrm{id},\mathbb{D})} & \left(\mathbf{Syn}_V, \mathbf{Syn}_{\mathcal{S}}, \underline{\nu}_{\mathbf{Syn}}\right) \times \left(\mathbf{Syn}_V^{\mathbf{tr}}, \mathbf{Syn}_{\mathcal{S}}^{\mathbf{tr}}, \underline{\nu}_{\mathbf{Syn}}^{\mathbf{tr}}\right) \\ \downarrow{\scriptstyle \overline{[\![-]\!]}_n} & & \downarrow{\scriptstyle [\![-]\!] \times [\![-]\!]_k} \\ \mathcal{U}_{rBV}\left(\mathbf{Sub}\left(\boldsymbol{\omega}\mathbf{Cpo} \downarrow G_n\right), \mathcal{P}_n\left(-\right)_\perp\right) & \xrightarrow[\mathcal{U}_{rBV}(\underline{\mathcal{L}}_n)]{} & \mathcal{U}_{rBV}\left(\boldsymbol{\omega}\mathbf{Cpo} \times \boldsymbol{\omega}\mathbf{Cpo}, \left(-\right)_\perp\right) \end{array} \tag{7.2}$$

7.1. AD correctness from logical relations. The first observation is that, indeed, our definitions give the desired predicate. More precisely, if $\left(h, \dot{h}\right) \in \mathcal{P}_n\left(\coprod_{j \in L} \overline{[\![\mathbf{real}]\!]}_n^{l_j}\right)_\perp$, *then* $h : \mathbb{R}^n \to \left(\coprod_{j \in L} \mathbb{R}^{l_j}\right)_\perp$ *is differentiable and* $\dot{h} = \overline{\mathfrak{d}^k\left(h\right)}$. By the definition of differentiable morphisms between coproducts of cartesian spaces, we have:

Theorem 7.1. *If, for each* $i \in \mathfrak{L}$*, the morphism* $\left(h, \dot{h}\right)$ *in* $\boldsymbol{\omega}\mathbf{Cpo} \times \boldsymbol{\omega}\mathbf{Cpo}$ *defines the morphism* (7.3) *in* $\mathbf{Sub}\left(\boldsymbol{\omega}\mathbf{Cpo} \downarrow G_{s_i}\right)$*, then* $h : \coprod_{r \in \mathfrak{L}} \mathbb{R}^{s_r} \to \left(\coprod_{j \in L} \mathbb{R}^{l_j}\right)_\perp$ *is differentiable and* $\dot{h} = \mathfrak{d}^k\left(h\right)$.

$$\mathtt{h} : \coprod_{r \in \mathfrak{L}} \overline{[\![\mathbf{real}]\!]}_{s_i}^{s_r} \to \mathcal{P}_{s_i}\left(\coprod_{j \in L} \overline{[\![\mathbf{real}]\!]}_{s_i}^{l_j}\right)_\perp \quad \left(h \circ \iota_{\mathbb{R}^{s_i}}, \dot{h} \circ \iota_{(\mathbb{R} \times \mathbb{R}^k)^{s_i}}\right) \tag{7.4}$$
$$\tag{7.3}$$

Proof: For each $i \in \mathfrak{L}$, we have that (7.4) defines a morphism $\mathtt{h}_i$ is a morphism from $\overline{[\![\mathbf{real}]\!]}_{s_i}^{s_i} \to \mathcal{P}_{s_i}\left(\coprod_{j \in L} \overline{[\![\mathbf{real}]\!]}_{s_i}^{l_j}\right)_\perp$ in $\mathbf{Sub}\left(\boldsymbol{\omega}\mathbf{Cpo} \downarrow G_{s_i}\right)$. This implies

$$\frac{(c \in \mathbb{R})}{\Gamma \vdash \underline{c} :: \mathbf{real}} \qquad \frac{\{\Gamma \vdash t_i : \mathbf{real}\}_{i=1}^n \quad (\mathrm{op} \in \mathrm{Op}_n)}{\Gamma \vdash \mathrm{op}(t_1, \ldots, t_n) : \mathbf{real}}$$

FIGURE 4.1. The type assignment rules for the primitive type of **real**.

list the full rules in Fig. B.1 in App. A.) We consider the standard CBV $\beta\eta$-equational theory (see [23]) for our language (which we list in Fig. B.3).

4.2. Target language. As mentioned above, our target language is a simple extension of our source language, introducing a new type **vect** that plays the role of (co)tangents. We do so by adding the following syntax, with the typing rules of Fig. 4.2 (for the full typing rules, see Fig. B.2 in App. B).

τ, σ, ρ	::=	types	|	**vect**	(co)tangent
	| …	as before			
v, w, u	::=	values	|	$t + s$	addition of vectors
	| …	as before	|	$t * s$	scalar multiplication
	| $\overline{e}_i$	i-th canonical element	|	$\mathfrak{h}_i t$	proj. handler
	| $\overline{0}$	zero			
t, s, r	::=	computations	|	$t + s$	addition of vectors
	| …	as before	|	$t * s$	scalar multiplication
	| $\overline{e}_i$	canonical element	|	$\mathfrak{h}_i t$	proj. handler
	| $\overline{0}$	zero			

$$\frac{(i \in \mathbb{N}^*)}{\Gamma \vdash \overline{e}_i : \mathbf{vect}} \qquad \frac{}{\Gamma \vdash \overline{0} : \mathbf{vect}} \qquad \frac{\Gamma \vdash t : \mathbf{vect} \quad \Gamma \vdash s : \mathbf{vect}}{\Gamma \vdash t + s : \mathbf{vect}}$$

$$\frac{\Gamma \vdash t : \mathbf{vect} \quad \Gamma \vdash s : \mathbf{real}}{\Gamma \vdash t * s : \mathbf{vect}} \qquad \frac{(i \in \mathbb{N}^*) \quad \Gamma \vdash t : \mathbf{vect}}{\Gamma \vdash \mathfrak{h}_i t : \mathbf{real}^i}$$

FIGURE 4.2. The type assignment rules for the **vect** type of cotangents.

We want $\left(\mathbf{vect}, +, *, \overline{0}\right)$ to implement the vector space $\left(\mathbb{R}^k, +, *, 0\right)$, for some $k \in \mathbb{N} \cup \{\infty\}$[§]. In this case: **(A)** $\overline{e}_i$ should implement the i-th element $e_i^k \in \mathbb{R}^k$ of the canonical basis if $k = \infty$ or if $i \leq k$, and $0 \in \mathbb{R}^k$ otherwise; **(B)** $\mathfrak{h}_i t$

[§] $\mathbb{R}^\infty$ is the vector space freely generated by the infinite set $\{e_i : i \in \mathbb{N}^*\}$. In other words, it is the infinity coproduct of $\mathbb{R}^i$. In order to implement it, one can use lists/dynamically sized arrays and pattern matching for the vector addition.

should implement $\mathfrak{p}_{k\to i} : \mathbb{R}^k \to \mathbb{R}^i$ which denotes the canonical projection if $i \leq k$ and the coprojection otherwise.

For short, we say that **vect** *implements the vector space* $\mathbb{R}^k$ to refer to the case above. For efficient implementations [29], the operational semantics should make clever use of the usual distributive law of the scalar multiplication over the vector addition (aka linear factoring [6]). Although this is negligible from our semantical viewpoint, we observe that this optimisation is semantically correct.

4.3. Languages as rCBV models. As stressed in 2.3, we have the *syntactic rCBV models* given by the freely generated $rCBV$ models on our source and target languages, respectively denoted herein by $rCBV$ models $\left(\mathbf{Syn}_V, \mathbf{Syn}_{\mathcal{S}}, \underline{\nu}_{\mathbf{Syn}}\right)$ and $\left(\mathbf{Syn}^{\mathbf{tr}}_V, \mathbf{Syn}^{\mathbf{tr}}_{\mathcal{S}}, \underline{\nu}^{\mathbf{tr}}_{\mathbf{Syn}}\right)$.

Theorem 4.1 (Universal property of syntax)**.** *Let* $(\mathcal{V}, \mathcal{T}, \underline{\nu})$ *be an rCBV model. Assume that* (4.3) *and* (4.4) *are given consistent assignments for each* $\mathrm{op} \in \mathrm{Op}$, $\underline{c}$ *and* $i \in \mathbb{N}^*$. *There are unique rCBV model morphisms* $H : \left(\mathbf{Syn}_V, \mathbf{Syn}_{\mathcal{S}}, \underline{\nu}_{\mathbf{Syn}}\right) \to (\mathcal{V}, \mathcal{T}, \underline{\nu})$ *respecting* (4.3) *and* $\mathcal{H} : \left(\mathbf{Syn}^{\mathbf{tr}}_V, \mathbf{Syn}^{\mathbf{tr}}_{\mathcal{S}}, \underline{\nu}^{\mathbf{tr}}_{\mathbf{Syn}}\right) \to (\mathcal{V}, \mathcal{T}, \underline{\nu})$, *where* $\mathcal{H}$ *extends* H *and respects* (4.4).

$$\left(H(\mathbf{real}) \in \mathsf{ob}\,\mathcal{V}, H(\mathbf{sign}), H(\underline{c}), H(\mathrm{op})\right) \tag{4.3}$$

$$\left(\mathcal{H}(\mathbf{vect}) \in \mathsf{ob}\,\mathcal{V}, \mathcal{H}(\overline{0}), \mathcal{H}\left(\mathfrak{h}_i\right), \mathcal{H}(+), \mathcal{H}(*)\right) \tag{4.4}$$

4.4. Dual numbers AD code transformation. We define our *automatic differentiation macro* by making use of the universal property of the source language. Let us fix, for all $n \in \mathbb{N}$, $\mathrm{op} \in \mathrm{Op}_n$, and $1 \leq i \leq n$, computations $x_1 : \mathbf{real}, \ldots, x_n : \mathbf{real} \vdash \partial_i \mathrm{op}(x_1, \ldots, x_n) : \mathbf{real}$, which represent the partial derivatives of op.

The assignment defined in Fig. 4.6 induces a unique $rCBV$ model morphism (4.5), which corresponds to the structure preserving macro $\mathcal{D}$ on the types and computations of our language for performing AD defined in Fig. 4.7.

$$\mathbb{D} : \left(\mathbf{Syn}_V, \mathbf{Syn}_{\mathcal{S}}, \underline{\nu}_{\mathbf{Syn}}\right) \to \left(\mathbf{Syn}^{\mathbf{tr}}_V, \mathbf{Syn}^{\mathbf{tr}}_{\mathcal{S}}, \underline{\nu}^{\mathbf{tr}}_{\mathbf{Syn}}\right). \tag{4.5}$$

5. Semantics of dual numbers AD

Dual numbers AD relies on the interleaving of the function with its derivative. Semantically, we do so by considering the interleaving morphism between discrete ω-cpos. We briefly establish the formal definitions in this section.

$$\in \ \mathbf{Sub}\left(\boldsymbol{\omega}\mathbf{Cpo} \downarrow G_n\right).$$

We define $\mathcal{P}_n\left(-\right)_\perp$ on $\mathbf{Sub}\left(\boldsymbol{\omega}\mathbf{Cpo} \downarrow G_n\right)$ by (6.3) where $\underline{\mathcal{P}_n\left(D, (C, C'), j\right)_\perp}$ is the union (6.4) with the full $\boldsymbol{\omega}\mathbf{Cpo}$-substructure of $G_n\left(\overline{(C)_\perp, (C')_\perp}\right)$ induced by the inclusion $\mathtt{j}_X$ which is defined by the components given in ($\mathfrak{c}$.1), ($\mathfrak{c}$.2), and ($\mathfrak{c}$.3).

$$\begin{aligned}\left(\alpha_0, \alpha_1 = \left(\beta_0 : U \to C, \beta_1 : \phi_{n,k}\left(U \times \left(\mathbb{R}^k\right)^n\right) \to C'\right)\right)\\ \mapsto \left(\overline{\beta_0} : \mathbb{R}^n \to (C)_\perp, \overline{\beta_1} : \left(\mathbb{R} \times \mathbb{R}^k\right)^n \to (C')_\perp\right)\end{aligned} \tag{6.2}$$

$$\mathcal{P}_n\left(D, (C, C'), j\right)_\perp \stackrel{\text{def}}{=} \left(\underline{\mathcal{P}_n\left(D, (C, C'), j\right)_\perp}, \left((C)_\perp, (C')_\perp\right), \mathtt{j}_X\right) \tag{6.3}$$

$$\{\perp\} \sqcup G_n(C, C') \sqcup \left(\coprod_{U \in \mathfrak{O}_n} \mathbf{Sub}\left(\boldsymbol{\omega}\mathbf{Cpo} \downarrow G_n\right)\left(\mathsf{Diff}_{(U,n)}, \left(D, (C, C'), j\right)\right)\right) \tag{6.4}$$

$\mathfrak{c}$.1 the inclusion $\{\perp\} \to G_n\left((C)_\perp, (C')_\perp\right)$ of the least morphism

$$\perp : \left(\mathbb{R}^n, \left(\mathbb{R} \times \mathbb{R}^k\right)^n\right) \to \left((C)_\perp, (C')_\perp\right)$$

in $\boldsymbol{\omega}\mathbf{Cpo} \times \boldsymbol{\omega}\mathbf{Cpo}\left(\left(\mathbb{R}^n, \left(\mathbb{R} \times \mathbb{R}^k\right)^n\right), \left((C)_\perp, (C')_\perp\right)\right)$;

$\mathfrak{c}$.2 the inclusion of the total functions $G_n\left(\eta_C, \eta_{C'}\right) : G_n\left(C, C'\right) \to G_n\left((C)_\perp, (C')_\perp\right)$;

$\mathfrak{c}$.3 for each $U \in \mathfrak{O}_n$, the injection

$$\mathbf{Sub}\left(\boldsymbol{\omega}\mathbf{Cpo} \downarrow G_n\right)\left(\mathsf{Diff}_{(U,n)}, \left(D, (C, C'), j\right)\right) \to G_n\left((C)_\perp, (C')_\perp\right)$$

defined by (6.2) where $\overline{\beta_0}$ and $\overline{\beta_1}$ are the respective corresponding canonical extensions.

By lifting the multiplication and unit of $\left(-\right)_\perp$, the definition above gives us a strong monad $\mathcal{P}_n\left(-\right)_\perp$ on $\mathbf{Sub}\left(\boldsymbol{\omega}\mathbf{Cpo} \downarrow G_n\right)$ that is a lifting of $\left(-\right)_\perp$ along the forgetful $\boldsymbol{\omega}\mathbf{Cpo}$-functor $\underline{\mathcal{L}}_n : \mathbf{Sub}\left(\boldsymbol{\omega}\mathbf{Cpo} \downarrow G_n\right) \to \boldsymbol{\omega}\mathbf{Cpo} \times \boldsymbol{\omega}\mathbf{Cpo}$. Moreover, it is clear that $\mathcal{P}_n\left(-\right)_\perp$ satisfies the conditions of 3.1. Therefore, by Theorem 3.2, $\left(\mathbf{Sub}\left(\boldsymbol{\omega}\mathbf{Cpo} \downarrow G_n\right), \mathcal{P}_n\left(-\right)_\perp\right)$ is an $rCBV$ $\boldsymbol{\omega}\mathbf{Cpo}$-pair, and $\underline{\mathcal{L}}_n$ yields an $rCBV$ $\boldsymbol{\omega}\mathbf{Cpo}$-pair morphism. We have now effectively given a logical relations reasoning principle for derivatives of partially defined functions.

7. Logical relations for "real" – kickstarting the LR proof

We can, now, establish the logical relations' assignment. By the observation on differentiability, it is clear that we want to assign **real** to the object (7.1) in $\mathbf{Sub}\left(\boldsymbol{\omega}\mathbf{Cpo} \downarrow G_n\right)$. While for each primitive operation $\mathrm{op} \in \mathrm{Op}_m$ of the syntax,[¶] we have that the pair $\left([\![op]\!], [\![\partial^k\left([\![\mathrm{op}]\!]\right)]\!]_k\right)$ defines a morphism

[¶] We consider, here, the constants and **sign** as well.

$$[\![\partial_j \mathrm{op}(y_1,\ldots,y_n)]\!] = \mathfrak{d}_j([\![\mathrm{op}]\!]) \quad (5.5) \qquad [\![\mathcal{D}(t)]\!]_k = \mathfrak{d}^k([\![t]\!]) \quad (5.6) \qquad [\![\mathbb{D}(\mathrm{op})]\!]_k = \mathfrak{d}^k([\![\mathrm{op}]\!]) \quad (5.7)$$

6. Subscone for partially defined differentiable functions

Henceforth, we assume that **vect** implements the vector space $\mathbb{R}^k$, and that (5.7) holds. We establish the categorical framework of our correctness proof. Since we already established the basic semantics (5.4), we start by establishing our subscone: more precisely, the family of base LR $\boldsymbol{\omega}$**Cpo**-functors satisfying the setting of Section 3.

The differentiability and the derivative of a total function $g : \mathbb{R}^m \to C$ (where C is some manifold) are fully characterized by the differentiability and the derivatives of $g \circ \alpha$ for differentiable maps $\mathbb{R}^n \to \mathbb{R}^m$ ($n \in \mathbb{N}$). More precisely, *a total function $g : \mathbb{R}^m \to C$ is differentiable and $\dot{g} = \mathfrak{D}^k g$ if and only if $g \circ \alpha$ is differentiable and $\dot{g} \circ \mathfrak{D}^k\alpha = \mathfrak{D}^k(g \circ \alpha)$ for any differentiable map $\alpha : \mathbb{R}^n \to \mathbb{R}^m$ (and any natural n).* One direction of this observation follows from the *chain-rule for derivatives*, while the other is trivial since we can take $\alpha = \mathrm{id}$.

This leads us to consider the family defined by (6.1), which clearly satisfies the setting of Section 3: hence, we have that, for each n, $\mathbf{Sub}(\boldsymbol{\omega}\mathbf{Cpo} \downarrow G_n)$ is an appropriate model for the values of our languages by Theorem 3.1.

$$\left(G_n\colon = \boldsymbol{\omega}\mathbf{Cpo} \times \boldsymbol{\omega}\mathbf{Cpo}\left(\left(\mathbb{R}^n, \left(\mathbb{R} \times \mathbb{R}^k\right)^n\right), (-,-)\right) : \boldsymbol{\omega}\mathbf{Cpo} \times \boldsymbol{\omega}\mathbf{Cpo} \to \boldsymbol{\omega}\mathbf{Cpo}\right)_{n\in\mathbb{N}} \quad (6.1)$$

We proceed to define a monad lifting satisfying the setting of 3.1 in order to end up with an *rCBV* $\boldsymbol{\omega}$**Cpo**-pair and, hence, an *rCBV* model. We do that informed of the property that we want to prove. We start by extending the observation about the characterization of differentiability to the case of partially defined functions: namely, $h : \mathbb{R}^{n_i} \to \left(\coprod_{j\in L} \mathbb{R}^{m_j}\right)_\perp$ is differentiable and $\dot{h} = \mathfrak{d}^k(h_i)$ if and only if: **(A)** the domain of definition of h is an open set $U_i \subseteq \mathbb{R}^{n_i}$; **(B)** for any differentiable map $\alpha : \mathbb{R}^n \to U_i$ and any $n \in \mathbb{N}$, $h_{U_i} \circ \alpha$ is differentiable and, denoting by $\underline{\dot{h}}$ the total function corresponding to $\dot{h}$, $\underline{\dot{h}} \circ \mathfrak{D}^k\alpha$ is well defined and equal to $\mathfrak{D}^k(h_{U_i} \circ \alpha)$.

Informed of this observation and in order to establish the underlying predicate above, we define, for each $U \in \mathfrak{O}_n$, where $\mathfrak{O}_n$ is the set of proper open non-empty subsets of the cartesian space $\mathbb{R}^n$, the object $\mathsf{Diff}_{(U,n)}$:

$$\mathsf{Diff}_{(U,n)} \stackrel{\text{def}}{=} \left(\left\{\left(g : \mathbb{R}^n \to U, \mathfrak{D}^k g\right) : g \text{ is differentiable}\right\}, \left(U, \phi_{n,k}\left(U \times \left(\mathbb{R}^k\right)^n\right)\right), \mathrm{incl.}\right)$$

$$\mathbb{D}(\mathbf{real}) \stackrel{\text{def}}{=} \mathbf{real} \times \mathbf{vect} \in \mathsf{ob}\,\mathbf{Syn}_V^{\mathbf{tr}}, \qquad \mathbb{D}(\underline{c}) \stackrel{\text{def}}{=} (\underline{c}, \underline{0}) \in \mathbf{Syn}_V^{\mathbf{tr}}(1, \mathbf{real} \times \mathbf{vect}),$$
$$\mathbb{D}(\mathrm{op}) \stackrel{\text{def}}{=} \overrightarrow{d}\,\mathrm{op}\,(y_1,\ldots,y_n) \in \mathbf{Syn}_V^{\mathbf{tr}}((\mathbf{real} \times \mathbf{vect})^n, \mathbf{Syn}_{\mathcal{S}}(\mathbf{real} \times \mathbf{vect})),$$
$$\mathbb{D}(\mathbf{sign}) \stackrel{\text{def}}{=} (\mathbf{sign} \circ \pi_1) \in \mathbf{Syn}_V^{\mathbf{tr}}(\mathbf{real} \times \mathbf{vect}, \mathbf{Syn}_{\mathcal{S}}(1 \sqcup 1)), \text{ where}$$

$$\begin{aligned}\overrightarrow{d}\,\mathrm{op}\,(y_1,\ldots,y_n) \stackrel{\text{def}}{=}\ & \mathbf{case}\, y_1\, \mathbf{of}\, \langle x_1, x_1'\rangle \to \ldots \to \mathbf{case}\, y_n\, \mathbf{of}\, \langle x_n, x_n'\rangle \to \\ & \mathbf{let}\, y' = \mathrm{op}(x_1,\ldots,x_n)\, \mathbf{in} \\ & \mathbf{let}\, z_1 = \partial_1\mathrm{op}(x_1,\ldots,x_n)\, \mathbf{in}\, \ldots \mathbf{let}\, z_n = \partial_n\mathrm{op}(x_1,\ldots,x_n)\, \mathbf{in} \\ & \langle y', x_1' * z_1 + \ldots + x_n' * z_n\rangle.\end{aligned}$$

FIGURE 4.6. AD assignment, for each primitive operation $\mathrm{op} \in \mathrm{Op}_n$ ($n \in \mathbb{N}$) and each constant $c \in \mathtt{R}$.

$$\mathcal{D}(\alpha) \stackrel{\text{def}}{=} \alpha \qquad \mathcal{D}(\mu\alpha.\tau) \stackrel{\text{def}}{=} \mu\alpha.\mathcal{D}(\tau)$$
$$\mathcal{D}(\mathbf{real}) \stackrel{\text{def}}{=} \mathbf{real} \times \mathbf{vect} \qquad \mathcal{D}(\mathbf{0}) \stackrel{\text{def}}{=} \mathbf{0}$$
$$\mathcal{D}(\mathbf{1}) \stackrel{\text{def}}{=} \mathbf{1} \qquad \mathcal{D}(\tau \sqcup \sigma) \stackrel{\text{def}}{=} \mathcal{D}(\tau) \sqcup \mathcal{D}(\sigma)$$
$$\mathcal{D}(\tau \to \sigma) \stackrel{\text{def}}{=} \mathcal{D}(\tau) \to \mathcal{D}(\sigma) \qquad \mathcal{D}(\tau \times \sigma) \stackrel{\text{def}}{=} \mathcal{D}(\tau) \times \mathcal{D}(\sigma)$$

$$\mathcal{D}(x) \stackrel{\text{def}}{=} x \qquad \mathcal{D}(\mathbf{let}\, x = t\, \mathbf{in}\, s) \stackrel{\text{def}}{=} \mathbf{let}\, x = \mathcal{D}(t)\, \mathbf{in}\, \mathcal{D}(s)$$
$$\mathcal{D}(\mathbf{case}\, r\, \mathbf{of}\, \{\,\}) \stackrel{\text{def}}{=} \mathbf{case}\, \mathcal{D}(r)\, \mathbf{of}\, \{\,\}$$
$$\mathcal{D}(\mathbf{inl}\, t) \stackrel{\text{def}}{=} \mathbf{inl}\, \mathcal{D}(t) \qquad \mathcal{D}(\mathbf{case}\, r\, \mathbf{of}\, \{\ \begin{array}{l}\mathbf{inl}\, x \to t \\ \mid\ \mathbf{inr}\, y \to s\end{array}\ \}) \stackrel{\text{def}}{=}$$
$$\mathcal{D}(\mathbf{inr}\, t) \stackrel{\text{def}}{=} \mathbf{inr}\, \mathcal{D}(t) \qquad \mathbf{case}\, \mathcal{D}(r)\, \mathbf{of}\, \{\ \begin{array}{l}\mathbf{inl}\, x \to \mathcal{D}(t) \\ \mid\ \mathbf{inr}\, y \to \mathcal{D}(s)\end{array}\ \}$$
$$\mathcal{D}(\langle\rangle) \stackrel{\text{def}}{=} \langle\rangle$$
$$\mathcal{D}(\langle t, s\rangle) \stackrel{\text{def}}{=} \langle \mathcal{D}(t), \mathcal{D}(s)\rangle \qquad \mathcal{D}(t\,r) \stackrel{\text{def}}{=} \mathcal{D}(t)\,\mathcal{D}(r)$$
$$\mathcal{D}(\lambda x.t) \stackrel{\text{def}}{=} \lambda x.\mathcal{D}(t)$$
$$\mathcal{D}(\mathbf{case}\, r\, \mathbf{of}\, \langle x, y\rangle \to t) \stackrel{\text{def}}{=} \mathbf{case}\, \mathcal{D}(r)\, \mathbf{of}\, \langle x, y\rangle \to \mathcal{D}(t)$$
$$\mathcal{D}(\mathbf{roll}\, t) \stackrel{\text{def}}{=} \mathbf{roll}\, \mathcal{D}(t) \qquad \mathcal{D}(\mathbf{case}\, t\, \mathbf{of}\, \mathbf{roll}\, x \to s) \stackrel{\text{def}}{=} \mathbf{case}\, \mathcal{D}(t)\, \mathbf{of}\, \mathbf{roll}\, x \to \mathcal{D}(s)$$

$$\begin{aligned}\mathcal{D}(\underline{c}) \stackrel{\text{def}}{=}\ & \langle \underline{c}, \overline{0}\rangle \\ \mathcal{D}(\mathrm{op}(r_1,\ldots,r_n)) \stackrel{\text{def}}{=}\ & \mathbf{case}\, \mathcal{D}(r_1)\, \mathbf{of}\, \langle x_1, x_1'\rangle \to \ldots \to \mathbf{case}\, \mathcal{D}(r_n)\, \mathbf{of}\, \langle x_n, x_n'\rangle \to \\ & \mathbf{let}\, y = \mathrm{op}(x_1,\ldots,x_n)\, \mathbf{in} \\ & \mathbf{let}\, z_1 = \partial_1\mathrm{op}(x_1,\ldots,x_n)\, \mathbf{in}\, \ldots \mathbf{let}\, z_n = \partial_n\mathrm{op}(x_1,\ldots,x_n)\, \mathbf{in} \\ & \langle y, x_1' * z_1 + \ldots + x_n' * z_n\rangle \\ \mathcal{D}(\mathbf{sign}\, r) \stackrel{\text{def}}{=}\ & \mathbf{sign}\,(\mathbf{fst}\, \mathcal{D}(r))\end{aligned}$$

FIGURE 4.7. AD macro $\mathcal{D}(-)$ defined on types and computations. All newly introduced variables are chosen to be fresh.

Henceforth, unless stated otherwise, the cartesian spaces $\mathbb{R}^n$ and its subspaces are endowed with the discrete $\boldsymbol{\omega}\mathbf{Cpo}$-structures, i.e. $x \leq y$ iff $x = y$, for all $x, y \in \mathbb{R}^n$.

Definition 5.1 (Interleaving function)**.** For each $(n, k) \in \mathbb{N} \times (\mathbb{N} \cup \{\infty\})$, denoting by $\mathbb{I}_n$ the set $\{1, \ldots, n\}$, we define the isomorphism (in $\boldsymbol{\omega}\mathbf{Cpo}$)

$$\begin{aligned} \phi_{n,k} : \quad \mathbb{R}^n \times \left(\mathbb{R}^k\right)^n &\to \left(\mathbb{R} \times \mathbb{R}^k\right)^n \\ \left((x_j)_{j\in\mathbb{I}_n}, (y_j)_{j\in\mathbb{I}_n}\right) &\mapsto (x_j, y_j)_{j\in\mathbb{I}_n}. \end{aligned} \tag{5.1}$$

For each open subset $U \subseteq \mathbb{R}^n$, we denote by $\phi^U_{n,k} : U \times \left(\mathbb{R}^k\right)^n \to \phi_{n,k}\left(U \times \left(\mathbb{R}^k\right)^n\right)$ the isomorphism obtained from restricting $\phi_{n,k}$.

Before we proceed to define the interleaved derivative of a total defined function, we have two remaining relevant remarks. Firstly, as opposed to the usual, we also care about the derivative of functions between (possibly infinite) coproducts of open subspaces of various dimensions. We extend the usual definition for this case in the obvious manner: it is just the corresponding definition for the free cocompletion of the category of connected spaces under coproducts. Put differently, we treat derivatives a local operations on functions – they can be computed restricted to each connected component of the input and glued together. The second remark is that we are particularly interested in a certain presentation of the derivative, via the transpose of the derivative (the natural semantical counterpart of reverse-mode AD). We present the precise definition.

Definition 5.2 (Vectorised derivative)**.** Let $g : U \to \coprod_{j\in L} V_j$ be a map where U is an open subset of $\mathbb{R}^n$, and, for each $i \in L$, V_i is an open subset of $\mathbb{R}^{m_i}$.

The map g is *differentiable* if, for any $i \in L$, $g^{-1}(V_i) = W_i$ is open in $\mathbb{R}^n$ and the restriction $g|_{W_i} : W_i \to V_i$ is differentiable w.r.t the submanifold structures $W_i \subseteq \mathbb{R}^n$ and $V_i \subseteq \mathbb{R}^{m_i}$. In this case, for each $k \in (\mathbb{N} \cup \{\infty\})$, we define the function $\mathfrak{D}^k g$, which we think of as the k-dimensional vectorised derivative:

$$\begin{aligned} \mathfrak{D}^k g : \ \phi_{n,k}\left(U \times \left(\mathbb{R}^k\right)^n\right) &\to \coprod_{j\in L}\left(\phi_{m_j,k}\left(V_i \times \left(\mathbb{R}^k\right)^{m_i}\right)\right) \\ z &\mapsto \iota_{m_j} \circ \phi^{V_j}_{m_j,k}\left(g(x), \tilde{w} \cdot g'(x)^t\right), \\ &\quad \text{whenever } \phi^{-1}_{n,k}(z) = (x, w) \in W_i \times \left(\mathbb{R}^k\right)^n \end{aligned} \tag{5.2}$$

in which $\tilde{w}$ is the linear transformation $\mathbb{R}^n \to \mathbb{R}^k$ corresponding to the vector w, $\cdot$ is the composition of linear transformations, ι_{m_i} is the obvious ith-coprojection of the coproduct (in the category $\boldsymbol{\omega}\mathbf{Cpo}$), and $g'(x)^t$ is the transpose of the derivative $g'(x) : \mathbb{R}^n \to \mathbb{R}^{m_i}$ of $g|_{W_i} : W_i \to V_i$ at $x \in U$.

We extend the definition above to the case of partially defined functions. More precisely, a partially defined function given by $h_i : \mathbb{R}^{n_i} \to \left(\coprod_{j\in L} \mathbb{R}^{m_j}\right)_\perp$ *is differentiable* if its domain of definition U_i is open in $\mathbb{R}^{n_i}$, and the corresponding total function defined in U_i is differentiable in the sense of Def. 5.2. Moreover, in this case, we define the *derivative* $\mathfrak{d}^k(h_i)$ by (5.3). Finally, a partially defined function $h : \coprod_{r\in K} \mathbb{R}^{n_r} \to \left(\coprod_{j\in L} \mathbb{R}^{m_j}\right)_\perp$ if every component $h_i := h \circ \iota_i$ is differentiable. In this case, the derivative $\mathfrak{d}^k(h)$ is defined by $\langle \mathfrak{d}^k(h_r)\rangle_{r\in K}$.

$$\begin{aligned} \mathfrak{d}^k(h_i) : \ \left(\mathbb{R} \times \mathbb{R}^k\right)^{n_i} &\to \left(\coprod_{j\in L}\left(\mathbb{R} \times \mathbb{R}^k\right)^{m_j}\right)_\perp \\ z &\mapsto \begin{cases} \mathfrak{D}^k h_i(z), & \text{if } z \in \phi_{n_i,k}\left(U_i \times \left(\mathbb{R}^k\right)^{n_i}\right) \subseteq \left(\mathbb{R} \times \mathbb{R}^k\right)^{n_i}; \\ \perp, & \text{otherwise.} \end{cases} \end{aligned} \tag{5.3}$$

5.1. Basic semantics for the languages. We give a concrete semantics for our language, interpreting it in the *rCBV* $\boldsymbol{\omega}\mathbf{Cpo}$-pair $(\boldsymbol{\omega}\mathbf{Cpo}, (-)_\perp)$: namely, by Theorem 4.1, we get a unique *rCBV* model morphism (5.4) defined by the assignment of **real** and each primitive operation $\mathbf{sign}$, $\underline{c}$, op with the corresponding intended semantics $\mathbb{R}$ (with the discrete $\boldsymbol{\omega}\mathbf{Cpo}$-structure), $[\![\mathbf{sign}]\!] := \mathsf{sign} : \mathbb{R} \to (\mathbf{1} \sqcup \mathbf{1})_\perp$, $[\![\underline{c}]\!] := \mathsf{c} : \mathbb{R} \to (\mathbf{1} \sqcup \mathbf{1})_\perp$, $[\![\mathrm{op}]\!] := f_{\mathrm{op}} : \mathbb{R}^n \to (\mathbb{R})_\perp$. Moreover, for each $k \in \mathbb{N} \cup \{\infty\}$, we can extend $[\![-]\!]$ into an *rCBV* model morphism $[\![-]\!]_k$ by interpreting $\left(\mathbf{vect}, +, *, \overline{0}\right)$ as the vector space $\left(\mathbb{R}^k, +, *, 0\right)$ it intends to implement, and the handlers as the appropriate (co)projections. The *rCBV* model morphism $[\![-]\!]_k$ defines the *k-semantics for the target language*. We can, then, consider the k-combined semantics of the product of our languages:

$$[\![-]\!] \times [\![-]\!]_k : \left(\mathbf{Syn}_V, \mathbf{Syn}_{\mathcal{S}}, \underline{\nu}_{\mathbf{Syn}}\right) \times \left(\mathbf{Syn}^{\mathbf{tr}}_V, \mathbf{Syn}^{\mathbf{tr}}_{\mathcal{S}}, \underline{\nu}^{\mathbf{tr}}_{\mathbf{Syn}}\right) \to \mathcal{U}_{\mathcal{BV}}\left(\boldsymbol{\omega}\mathbf{Cpo}^2, (-)_\perp\right) \tag{5.4}$$

We want to prove the semantical correctness of our macro $\mathcal{D}$, defined by $\mathbb{D}$: namely, denoting by $\mathfrak{d}_j(f)$ the usual j-th partial derivative of f, assuming that (5.5) holds for any primitive op $\in$ Op, we want to get (5.6) for any t between objects corresponding to data types in our language. It should be noted that, from (5.5), we already get (5.7).