A Simple Differentiable Programming Language

MARTÍN ABADI, Google Research, United States

GORDON D. PLOTKIN, Google Research, United States

Automatic differentiation plays a prominent role in scientific computing and in modern machine learning, often in the context of powerful programming systems. The relation of the various embodiments of automatic differentiation to the mathematical notion of derivative is not always entirely clear—discrepancies can arise, sometimes inadvertently. In order to study automatic differentiation in such programming contexts, we define a small but expressive programming language that includes a construct for reverse-mode differentiation. We give operational and denotational semantics for this language. The operational semantics employs popular implementation techniques, while the denotational semantics employs notions of differentiation familiar from real analysis. We establish that these semantics coincide.

CCS Concepts: • **Theory of computation → Denotational semantics**; **Operational semantics**; • **Software and its engineering → Domain specific languages**; • **Computing methodologies → Machine learning**.

Additional Key Words and Phrases: automatic differentiation, differentiable programming.

1 INTRODUCTION

Automatic differentiation is a set of techniques for calculating the derivatives of functions described by computer programs (e.g., [Baydin et al. 2018; Griewank 2000; Hascoët and Pascual 2013; Pearlmutter and Siskind 2008]). These techniques are not required to produce symbolic representations for derivatives as in classic symbolic differentiation; on the other hand, neither do they employ finite-difference approximation methods common in numerical differentiation. Instead, they rely on the chain rule from calculus to obtain the desired derivatives from those of the programs's basic operations. Thus, automatic differentiation is at the intersection of calculus and programming. However, the programs of interest are more than chains of operations: they may include control-flow constructs, data structures, and computational effects (e.g., side-effects or exceptions). Calculus does not provide an immediate justification for the treatment of such programming-language features.

In the present work we help bridge the gap between rules for automatic differentiation in expressive programming languages and their mathematical justification in terms of denotational semantics. Specifically, we consider automatic differentiation from a programming-language perspective by defining and studying a small but powerful functional first-order language. The language has conditionals and recursively defined functions (from which loops can be constructed), but only rudimentary data structures. Additionally, it contains a construct for reverse-mode differentiation,

explained in detail below. Our language is thus inspired by modern systems for machine learning, which include standard programming constructs and support reverse-mode differentiation. Reverse-mode differentiation permits the computation of gradients, forward-mode derivatives, and more. Indeed as our differentiation construct is a language primitive, differentiations can be nested within differentiations, allowing the computation of higher-order derivatives.

In the setting of a language such as ours, we can consider some common approaches to implementing differentiation:

- One approach relies on code transformation, whether on source code or intermediate representations. For example, for the derivative of a conditional expression if B then M_1 else M_2, it would output if B then N_1 else N_2, where N_1 and N_2 are the derivatives of M_1 and M_2 respectively. This approach is employed, for instance, in Theano [Bergstra et al. 2010], TensorFlow 1.0 [Abadi et al. 2016a; Yu et al. 2018], and Tangent [van Merrienboer et al. 2018].
- Another approach relies on tracing, typically eliminating control structures to produce a simpler form of code, which we call an *execution trace*, that can more easily be differentiated. For example, to produce the derivative of if B then M_1 else M_2, tracing would evaluate the conditional and produce a trace of the branch taken. Execution traces correspond to graphs of basic operations, and can be taken to be sequences of elementary assignments or else functional programs in A-normal form. Their derivatives can be calculated by applying the chain rule to those basic operations, perhaps via a code transformation (but now of a much simpler kind). Tracing may also record some intermediate values in an *evaluation trace*, to reduce, or eliminate, the need for recomputation.[1]

  This approach thereby conveniently avoids the problem of defining code transformations for conditionals and many other language constructs. It can also be implemented efficiently, sometimes in part with JIT compilation. For these reasons, trace-based differentiation is of growing importance. It is employed, for instance, in Autograd [Maclaurin et al. 2015], TensorFlow Eager Mode [Agrawal et al. 2019], Chainer [Tokui et al. 2015], PyTorch [Paszke et al. 2019], and JAX [Frostig et al. 2018][2].

We therefore focus on trace-based differentiation, and give our language an operational semantics using the trace-based approach. To do so, we define a sublanguage of execution trace terms (called simply trace terms below). These have no conditionals, function definitions or calls, or reverse-mode differentiations. They do have local definitions, corresponding to fanout in the graphs, but may not be in A-normal form. Tracing is modeled by a new kind of evaluation, called symbolic evaluation. This uses an environment for the free variables of a term to remove conditionals and function calls. Function derivatives at a given value are evaluated in three stages: first, the function is traced at that value; next, the resulting trace term is symbolically differentiated (largely just using the chain rule), resulting in another such trace term; and, finally, that term is evaluated.

We do not account for some of the optimizations used in practice. Doing so would have been a more complicated enterprise, possibly with more arbitrary choices tied to implementation details, and we wished to get a more straightforward formalization working first.

From a mathematical perspective, both approaches to implementing differentiation pose correctness problems. In particular, functions defined using conditionals need not be continuous, let alone

---

[1]Terminology in the automatic differentiation literature varies. Here we follow [Griewank 2000] for evaluation traces. Our execution traces are, perhaps in somewhat different manifestations, variously termed Wengert lists or tapes or evaluation traces [Baydin et al. 2018; Pearlmutter and Siskind 2008]. They can also be seen as combinations of the operation and index traces of [Griewank 2000].

[2]See https://www.sysml.cc/doc/146.pdf.

differentiable. Consider, for example, the following definition

$$f(x\text{:real})\text{:real} \;=\; \text{if } (x < 0) \text{ then } 0 \text{ else } x$$

of the popular ReLU function [Goodfellow et al. 2016]. This function is not differentiable at 0. Further, changing the function body to if $(x < 0)$ then 0 else 1 yields a non-continuous function. What is more, both approaches can produce wrong answers even for differentiable functions! Consider, for example, the following definition of the identity function on the reals:

$$g(x\text{:real})\text{:real} \;=\; \text{if } (x = 0) \text{ then } 0 \text{ else } x$$

The derivative of this function at $x = 0$ is 1. However, differentiation "by branches" (whether by code transformation or tracing) would produce the wrong answer, 0.

In order to capture the mathematical perspective, in addition to its operational semantics we give our language a denotational semantics. This semantics is based on classical notions of differentiation from real analysis (see, for example, [Trench 2003]). That theory concerns multivariate functions on the reals defined on open domains, i.e., partial such functions with open domains of definition. In our semantics, we make use of those that are smooth (that is, those that can be partially differentiated any number of times). A particularly pleasing aspect of this mathematical development is how well domain theory (needed to account for recursion) interacts with differentiation.

Partiality is necessary, as for any language with general recursion, but it also gives us useful flexibility in connection with differentiation. For example, let $\dot{<}$ be the approximation to $<$ which is equal to it except on the diagonal (i.e., where both arguments are equal) where it is undefined. Then

$$\dot{f}(x\text{:real})\text{:real} \;=\; \text{if } (x \mathbin{\dot{<}} 0) \text{ then } 0 \text{ else } x$$

defines an approximation to ReLU which is undefined at 0. The approximation to $<$ is (unlike $<$) continuous (i.e., the pre-images of true and false are open sets), and the approximation to ReLU is differentiable wherever it is defined. Therefore, we design the semantics of our language so that it forbids functions such as f but allows related approximations such as $\dot{f}$. An interesting question is how satisfactory an idealization this is of programming practice (which in any case works with approximate reals). We return to this point in the final section.

Proceeding in this way, we obtain adequacy theorems (i.e., operational soundness and completeness theorems) connecting the operational semantics of our language with a denotational semantics based on the classical theory of differentiation of partially defined multivariate real functions. Our theorems apply not only to conditional expressions but to the full language.

In sum, the main contributions of this paper are: (1) a first-order language with conditionals, recursive function definitions, and a reverse-mode differentiation construct; (2) an operational semantics that models one form of trace-based differentiation; (3) a denotational semantics based on standard mathematical notions from real analysis and domain theory; and (4) theorems that show that the two semantics coincide, i.e., the derivatives computed by the operational semantics are indeed the correct derivatives in a mathematical sense. Beyond the specifics of these results, this paper aims to give some evidence of the relevance of ideas and techniques from the programming-languages literature for programming systems that include automatic differentiation, such as current systems for machine learning.

Additional context. While traditionally associated with scientific computing, automatic differentiation is now a central component of many modern machine learning systems, and those for deep learning in particular [Goodfellow et al. 2016]. These systems often employ automatic differentiation to compute the gradients of "loss functions" with respect to parameters, such as neural network weights. Loss functions measure the error resulting from particular values for the parameters. For example, when a machine-learning model is trained with a dataset consisting of

pairs $(x_0, y_0), \ldots, (x_{n-1}, y_{n-1})$, aiming to learn a function that maps the x_i's to the y_i's, the loss function may be the distance between the y_i's and the values the model predicts when presented with the x_i's. By applying gradient descent to adjust the parameters, this error can be reduced, until convergence or (more commonly) until the error is tolerable enough. This simple approach has proven remarkably effective: it is at the core of many recent successes of machine learning in a variety of domains.

Whereas gradients are for functions of type $\mathsf{real}^n \to \mathsf{real}$, for $n \geq 0$, treating the more general functions of type $\mathsf{real}^n \to \mathsf{real}^m$, for $n, m \geq 0$, works better with function composition, and with the composite structures such as tensors of reals used in deep learning. The literature contains two basic "modes" for differentiating such functions. Forward-mode extends the computation of the function, step by step, with the computation of derivatives; it can be seen as evaluating the function on dual numbers of the form $v + \dot{v}\varepsilon$ where ε is nilpotent. In contrast, reverse-mode propagates derivatives backwards from each output, typically after the computation of the function. Reverse-mode differentiation is often preferred because of its superior efficiency for functions of type $\mathsf{real}^n \to \mathsf{real}^m$ with $m \ll n$. In particular, systems for machine learning, which often deal with loss functions for which $m = 1$, generally rely on reverse-mode differentiation. We refer the reader to the useful recent survey [Baydin et al. 2018] for additional background on these two modes of differentiation; it also discusses the use of higher-order differentiation.

Applications to machine learning are our main motivation. Accordingly, our language is loosely inspired by systems for machine learning, and the implementation strategies that we consider are ones of current interest there. We also de-emphasize some concerns (e.g., numerical stability) that, at present, seem to play a larger role in scientific computing than in machine learning. As noted in [Baydin et al. 2016], the machine learning community has developed a mindset and a body of techniques distinct from those traditional in automatic differentiation.

The literature on scientific computing has addressed the correctness problem for conditionals [Beck and Fischer 1994; Fischer 2001], although not in the context of a formally defined programming language. In [Mayero 2002] a formal proof of correctness was given in Coq [Bertot and Castéran 2013] for an algorithm for the automatic differentiation of straight-line sequences of Fortran assignments. Closer to machine learning, [Selsam et al. 2017] consider a stochastic graphical formalism where the nodes are random variables, and use the Lean theorem prover [de Moura et al. 2015] to establish the correctness of stochastic backpropagation. However, overall, the literature does not seem to contain semantics and theorems for a language of the kind we consider here.

Our work is also related to important papers by Ehrhard, Regnier, et al. [Ehrhard and Regnier 2003], and by Di Gianantonio and Edalat [Di Gianantonio and Edalat 2013]. Ehrhard and Regnier introduce the differential λ-calculus; this is a simply-typed higher-order λ-calculus with a forward-mode differentiation construct which can be applied to functions of any type. It can be modeled using the category of convenient vector spaces and smooth functions between them (see [Blute et al. 2010; Kriegl and Michor 1997]). Ehrhard and Regnier do not give an operational semantics but they do give rules for symbolic differentiation and it should not be too difficult to use them to give an operational semantics. However their language with its convenient vector space semantics only supports total functions. It therefore cannot be extended to include recursive function definitions or conditionals (even with total predicates, as continuous functions from $\mathbb{R}^n$ to the booleans are constant). Di Gianantonio and Edalat prove adequacy theorems for their language, as do we, but their work differs from ours in several respects. In particular, their language has first-order forward-mode but no reverse-mode differentiation: our language effectively supports both, and at all orders. On the other hand, their language allows recursively-defined higher-order functions and accommodates functions, such as the ReLU function, which are differentiable in only a weaker sense. As far as we know, no other work on differentiable programming languages (e.g., [Elliott

2018; Manzyuk 2012; Pearlmutter and Siskind 2008; Shaikhha et al. 2018; Wang et al. 2018]) gives operational and denotational semantics and proves adequacy theorems. Further afield, there is a good deal of work in the categorical literature on categories equipped with differential structure, for example [Blute et al. 2009; Bucciarelli et al. 2010].

Outline. Section 2 defines our language. Section 3 gives it an operational semantics with rules for symbolically evaluating general terms to trace terms, and for symbolically differentiating these terms. Sections 4 and 5 cover the needed mathematical material and the denotational semantics. Sections 6 establishes the correspondence between operational and denotational semantics. Section 7 concludes with discussion and some suggestions for future work.

2 A SIMPLE LANGUAGE

The *types* $S, T, U, \ldots$ of our language are given by the grammar:

$$T \quad ::= \quad \mathtt{real} \mid \mathtt{unit} \mid T \times U$$

We will make use of iterated products $T_0 \times \ldots \times T_{n-1}$, defined to be $\mathtt{unit}$ when $n = 0$, T_0 when $n = 1$, and, recursively, $(T_0 \times \ldots \times T_{n-1}) \times T_n$, when $n > 1$; we write $\mathtt{real}^n$ for the n-fold iterated product of $\mathtt{real}$. Note that this type system includes the types of tensors (multidimensional arrays) of a given shape: the type of tensors of shape $(d_0, \ldots, d_{n-1})$ is the iterated product $\mathtt{real}^{d_0} \times \ldots \times \mathtt{real}^{d_{n-1}}$. The *terms* $L, M, N, P, \ldots$ and *boolean terms* B of the language are built from *operation symbols* op $\in$ Op and *predicate symbols* pred $\in$ Pred. An example operation symbol could be $DProd_n$ for dot product of vectors of dimension n (for $n \in \mathbb{N}$); an example predicate symbol could be $\dot{<}$.

The terms are given by the following grammar, where x and f range over disjoint countably infinite alphabets of ordinary and function variables, respectively. We assume available a standard ordering of the function variables.

$$
\begin{aligned}
M \quad ::= \quad & x \mid r \ (r \in \mathbb{R}) \mid M + N \mid \mathrm{op}(M) \mid \\
& \mathtt{let}\ x : T\ =\ M\ \mathtt{in}\ N \mid \\
& * \mid \langle M, N \rangle_{T,U} \mid \mathsf{fst}_{T,U}(M) \mid \mathsf{snd}_{T,U}(M) \mid \\
& \mathtt{if}\ B\ \mathtt{then}\ M\ \mathtt{else}\ N \mid \\
& \mathtt{letrec}\ f(x : T) : U\ =\ M\ \mathtt{in}\ N \mid \\
& f(M) \mid \\
& M.\mathsf{rd}_L(x : T.\ N)
\end{aligned}
$$

$$B \quad ::= \quad \mathrm{pred}(M) \mid \mathtt{true} \mid \mathtt{false}$$

These constructs are all fairly standard, except for rd, which is for reverse-mode differentiation, and which we explain below. We treat addition separately from the operations to underline the fact that the commutative monoid it forms, together with zero, is basic for differentiation. For example, the rules for symbolic differentiation given below make essential use of this structure, but do not need any more of the available vector space structure.

Note the type subscripts on pairing and projection terms. Below, we rely on these subscripts for symbolic differentiation. In practice they could, if needed, be added when type-checking.

The sets $\mathrm{FV}(M)$ and $\mathrm{FFV}(M)$ of *free ordinary variables* and *free function variables* of a term M are understood as usual (and similarly for boolean terms). As is also usual, we do not distinguish α-equivalent terms (or boolean terms).

The useful abbreviation

$$\mathtt{let}\ \langle x_0 : T_0, \ldots, x_{n-1} : T_{n-1} \rangle\ =\ M\ \mathtt{in}\ N$$

provides an elimination construct for iterated products. When $n = 0$ this is

$$\texttt{let } x\texttt{:unit } = \ M \texttt{ in } N$$

where $x \notin \text{FV}(N)$; when $n = 1$ it is the above let construct; otherwise, it is defined recursively by:

$$\texttt{let } \langle x_0\texttt{:}T_0, \ldots, x_n\texttt{:}T_n \rangle \ = \ M \texttt{ in } N = \ \begin{array}{l} \texttt{let } z\texttt{:}(T_0 \times \ldots \times T_n) \ = \ M \texttt{ in} \\ \texttt{let } \langle x_0\texttt{:}T_0, \ldots, x_{n-1}\texttt{:}T_{n-1} \rangle \ = \ \texttt{fst}_{T_0 \times \ldots \times T_{n-1}, T_n}(z) \texttt{ in} \\ \texttt{let } x_n\texttt{:}T_n \ = \ \texttt{snd}_{T_0 \times \ldots \times T_{n-1}, T_n}(z) \texttt{ in } N \end{array}$$

(where z is chosen not free in N).

We have zero and addition only at type real. At other types we proceed inductively:

$$0_{\texttt{real}} = 0 \qquad 0_{\texttt{unit}} = * \qquad 0_{T \times U} = \langle 0_T, 0_U \rangle$$

and:

$$M +_{\texttt{unit}} N \quad = \quad \begin{array}{l} \texttt{let } x\texttt{:unit } = \ M \texttt{ in} \\ \texttt{let } y\texttt{:unit } = \ N \texttt{ in } * \end{array}$$

$$M +_{T \times U} N \quad = \quad \begin{array}{l} \texttt{let } \langle x_1\texttt{:}T, x_2\texttt{:}U \rangle \ = \ M \texttt{ in} \\ \texttt{let } \langle y_1\texttt{:}T, y_2\texttt{:}U \rangle \ = \ N \texttt{ in} \\ \langle x_1 +_T y_1, x_2 +_U y_2 \rangle \end{array}$$

Skating over the difference between terms and their denotations, $M.\texttt{rd}_L(x\texttt{:}T.\,N)$ is the reverse-mode derivative at $L : T$, evaluated at $M : U$, of the function f such that $f(x : T) : U = N$. Reverse-mode differentiation includes gradients as a special case. When $T = \texttt{real}^n$ and $U = \texttt{real}$, the gradient of f at L is given by:

$$\texttt{grad}_L(x\texttt{:real}^n.\,N) \ = \ 1.\texttt{rd}_L(x\texttt{:real}^n.\,N)$$

For definitions of gradients, Jacobians, and derivatives see Section 4.1 below, particularly equations (1), (2), (3), and (4). More generally, for an introduction to real analysis including vector-valued functions of several variables and their differentials and Jacobians, see, for example, [Trench 2003].

As in [Christianson 2012], and as validated by equation (6) below, forward-mode differentiation can be defined using nested reverse-mode differentiation. We can set:

$$M.\texttt{fd}_{U,L}(x\texttt{:}T.\,N) \ = \ M.\texttt{rd}_{0_U}(y\texttt{:}U.\,y.\texttt{rd}_L(x\texttt{:}T.\,N))$$

So our language effectively also has forward-mode differentiation.

Function definitions can be recursive. Indeed a function can even be defined in terms of its own derivative: in a recursive function definition $\texttt{letrec } f(x\texttt{:}T)\texttt{:}U \ = \ M \texttt{ in } N$, the language allows occurrences of f within the term N' in a sub-term $M'.\texttt{rd}_{L'}(y : T'.\,N')$ of M. This generality may be useful—examples have arisen in the context of Autograd [Maclaurin et al. 2015][3]. Pleasantly, both our operational and denotational semantics accommodate it without special complications. When we define a function without recursion, we may abbreviate letrec to let.

Turning to typing, operation and predicate symbols have given arities, written $\texttt{op}\texttt{:}T \rightarrow U$ and $\texttt{pred}\texttt{:}T$; we write $\text{Op}_{T,U}$ for the set of operation symbols of arity $T \rightarrow U$. For example, we would have $DProd_n\texttt{:real}^n \times \texttt{real}^n \rightarrow \texttt{real}$ and $\dot{<}\texttt{:real}^2$. Figure 1 gives typing rules for sequents

$$\Phi \mid \Gamma \vdash M\texttt{:}T \qquad \Phi \mid \Gamma \vdash B$$

where *(type) environments* Γ have the form

$$x_0\texttt{:}T_0, \ldots, x_{n-1}\texttt{:}T_{n-1}$$

---

[3]See https://dougalmaclaurin.com/talk.pdf

(x_i all different) and where *function (type) environments* Φ have the form

$$f_0 : T_0 \to U_0, \ldots, f_{n-1} : T_{n-1} \to U_{n-1}$$

(f_i all different). We adopt the usual overwriting notations $\Gamma[\Gamma']$ and $\Phi[\Phi']$ for type environments.

$$\Phi \mid \Gamma \vdash x : T \quad (x : T \in \Gamma) \qquad \Phi \mid \Gamma \vdash r : \mathtt{real} \quad (r \in \mathbb{R})$$

$$\frac{\Phi \mid \Gamma \vdash M : \mathtt{real} \qquad \Phi \mid \Gamma \vdash N : \mathtt{real}}{\Phi \mid \Gamma \vdash M + N : \mathtt{real}}$$

$$\frac{\Phi \mid \Gamma \vdash M : T}{\Phi \mid \Gamma \vdash \mathtt{op}(M) : U} \quad (\mathtt{op} : T \to U)$$

$$\frac{\Phi \mid \Gamma \vdash M : T \qquad \Phi \mid \Gamma[x : T] \vdash N : U}{\Phi \mid \Gamma \vdash \mathtt{let}\ x : T = M\ \mathtt{in}\ N : U}$$

$$\Phi \mid \Gamma \vdash * : \mathtt{unit} \qquad \frac{\Phi \mid \Gamma \vdash M : T \qquad \Phi \mid \Gamma \vdash N : U}{\Phi \mid \Gamma \vdash \langle M, N \rangle_{T,U} : T \times U}$$

$$\frac{\Phi \mid \Gamma \vdash M : T \times U}{\Phi \mid \Gamma \vdash \mathtt{fst}_{T,U}(M) : T} \qquad \frac{\Phi \mid \Gamma \vdash M : T \times U}{\Phi \mid \Gamma \vdash \mathtt{snd}_{T,U}(M) : U}$$

$$\frac{\Phi \mid \Gamma \vdash B \qquad \Phi \mid \Gamma \vdash M : T \qquad \Phi \mid \Gamma \vdash N : T}{\Phi \mid \Gamma \vdash \mathtt{if}\ B\ \mathtt{then}\ M\ \mathtt{else}\ N : T}$$

$$\frac{\Phi[f : T \to U] \mid x : T \vdash M : U \qquad \Phi[f : T \to U] \mid \Gamma \vdash N : S}{\Phi \mid \Gamma \vdash \mathtt{letrec}\ f(x : T) : U = M\ \mathtt{in}\ N : S}$$

$$\frac{\Phi \mid \Gamma \vdash M : T}{\Phi \mid \Gamma \vdash f(M) : U} \quad (f : T \to U \in \Phi)$$

$$\frac{\Phi \mid \Gamma[x : T] \vdash N : U \qquad \Phi \mid \Gamma \vdash L : T \qquad \Phi \mid \Gamma \vdash M : U}{\Phi \mid \Gamma \vdash M.\mathtt{rd}_L(x : T.\ N) : T}$$

$$\Phi \mid \Gamma \vdash \mathtt{true} \qquad \Phi \mid \Gamma \vdash \mathtt{false} \qquad \frac{\Phi \mid \Gamma \vdash M : T}{\Phi \mid \Gamma \vdash \mathtt{pred}(M)} \quad (\mathtt{pred} : T)$$

Fig. 1. Typing rules

The typing rule for function definitions forbids any global variable occurrences (i.e., free variables in function definitions). This restriction involves no loss in expressiveness: as in lambda lifting, one can just add any global variables to a function's parameters, and then apply the function to the global variables wherever it is called. The restriction enabled us to prove part (2) of Theorem 6.2 (below), but we conjecture it is not needed.

Our various abbreviations have natural admissible typing rules:

$$\frac{\Phi \mid \Gamma \vdash M : T_0 \times \ldots \times T_{n-1} \qquad \Phi \mid \Gamma[x_0 : T_0, \ldots, x_{n-1} : T_{n-1}] \vdash N : U}{\Phi \mid \Gamma \vdash \mathtt{let}\ \langle x_0 : T_0, \ldots, x_{n-1} : T_{n-1} \rangle = M\ \mathtt{in}\ N : U}$$

$$\frac{\Phi \mid \Gamma \vdash N : \mathtt{real}^n \qquad \Phi \mid \Gamma[x : \mathtt{real}^n] \vdash M : \mathtt{real}}{\Phi \mid \Gamma \vdash \mathtt{grad}_N(x : \mathtt{real}^n.\ M) : \mathtt{real}^n}$$

$$\frac{\Phi \mid \Gamma[x : T] \vdash N : U \qquad \Phi \mid \Gamma \vdash L : T \qquad \Phi \mid \Gamma \vdash M : T}{\Phi \mid \Gamma \vdash M.\mathtt{fd}_{U,L}(x : T.\ N) : U}$$

$$\Phi \mid \Gamma \vdash 0_T : T \qquad \frac{\Phi \mid \Gamma \vdash M : T \qquad \Phi \mid \Gamma \vdash N : T}{\Phi \mid \Gamma \vdash M +_T N : T}$$

We may write $\Gamma \vdash M : T$ (or $\vdash M : T$) instead of $\Phi \mid \Gamma \vdash M : T$ if M has no free ordinary (or function) variables (and similarly for boolean terms). Typing is unique: for any Γ, Φ, and M there is at most one type T such that $\Phi \mid \Gamma \vdash M : T$ holds.

As an example, we use our language to program a miniature version of an algorithm for training a machine learning model by gradient descent, loosely based on [Goodfellow et al. 2016, Algorithm 8.1]. In such training, one often starts with an untrained model, which is a function from inputs (for example, images) and parameter values to output "predictions" (for example, image labels). Relying on a dataset of input/output pairs, one then picks values of the parameters by gradient descent, as indicated in the Introduction. In our miniature version, we treat inputs, parameter values, and outputs as reals, and we assume that the training data consists of only one fixed input/output pair (a, b). We also assume that we have real constants w_0 (for the initial value for gradient descent), *rate* (for the learning rate, which is fixed) and *maxLoss* (for the desired maximum loss on the dataset), and the infix predicate symbol $\dot{<}$. We can then define the trained model from the untrained model and a loss function as follows:

```
let myTrainedModel(x:real):real =
 let currentLoss(w:real):real = myLoss(⟨b, myUntrainedModel(⟨a, w⟩)⟩) in
 let gradLoss(w:real):real = grad_w(w′:real.currentLoss(w′)) in
 letrec descend(w:real):real =
     if currentLoss(w) ċ c then w else descend(w − rate ∗ gradLoss(w)) in
 myUntrainedModel(⟨x, descend(w_0)⟩)
in ...
```

The example above is typical of what can be expressed in our language, and many variants of machine learning techniques that rely on gradient descent (e.g., of the kind presented in [Goodfellow et al. 2016], and commonly used in systems like TensorFlow) are in scope as well. For instance, there is no difficulty in expressing optimization with momentum, or differentially private stochastic gradient descent (e.g., [Abadi et al. 2016b; Song et al. 2013]). Probabilistic choice may be treated via random number generators, as is done in practice. Architectures that rely on convolutions or RNN cells can be expressed, even conveniently, with a suitable choice of primitives.

3 OPERATIONAL SEMANTICS

We give a big-step operational semantics, specified with Felleisen and Friedman's method [Felleisen and Friedman 1987] using evaluation contexts and redexes. Other styles of operational semantics accommodating differentiation are surely also possible.

Terms and boolean terms are *(ordinarily) evaluated* to closed *values* and (necessarily) closed *boolean values*. The most original aspect of our operational semantics concerns the evaluation of differential terms; this is based on the trace-based approach outlined in the Introduction, and uses a second mode of evaluation: *symbolic evaluation*.

The core idea is that to evaluate a differential term

$$M.\mathsf{rd}_L(x : T.\, N)$$

one first evaluates L and M, and then performs differentiation before evaluating further. There are two differentiation stages. First, using the closed value V of L for the *differentiation variable x*, N is *symbolically evaluated* to a trace term C, thereby removing all control constructs from N, but possibly keeping the variable x free in C, as the derivative may well depend on it. For example, when N is if $x \dot{<} 0$ then 0 else x, the value V allows the guard of the conditional to be evaluated, but

the occurrence of x in the else branch is not replaced by V. Second, C is *symbolically differentiated* at V with respect to x.

However, this idea is not enough by itself as the differential term may occur inside yet another differential term. One therefore also needs to be able to symbolically evaluate the differential term. That is done much as in the case of ordinary evaluation, but now symbolically evaluating redexes in L and M until one is left with the problem of symbolically evaluating a term of the form

$$W.\mathsf{rd}_V(x{:}T.\,N)$$

where V and W are values that may contain free variables. One then proceeds as above, symbolically evaluating N (now using the closed value V' of V) and then performing symbolic differentiation. As there is some duplication between these two symbolic and ordinary evaluation processes, our rule for ordinarily evaluating a differential term is designed, when executed, to first symbolically evaluate the term, and then ordinarily evaluate the resulting trace term.

The need to keep track of differentiation variables and their values for symbolic evaluation leads us to use value environments for ordinary variables. It is convenient to also use them for ordinary evaluation and to use function environments for function variables for both modes of evaluation.

Values $V, W, X, \ldots$ are terms given by the grammar:

$$V \quad ::= \quad x \mid r \ (r \in \mathbb{R}) \mid * \mid \langle V, W \rangle_{T,U}$$

Note that, as indicated above, values may have free variables for the purposes of differentiation. Boolean values V_{bool} are boolean terms given by:

$$V_{\mathsf{bool}} ::= \mathsf{true} \mid \mathsf{false}$$

Closed values have unique types $\vdash V{:}T_V$; the set of closed values of type T is Val_T; and the set of boolean values is $\mathrm{Val}_{\mathsf{bool}}$. We assume available operation and predicate symbol *evaluation* functions

$$\mathrm{ev} : \mathrm{Op}_{T,U} \times \mathrm{Val}_T \rightharpoonup \mathrm{Val}_U \qquad \mathrm{bev} : \mathrm{Pred}_T \times \mathrm{Val}_T \rightharpoonup \mathrm{Val}_{\mathsf{bool}}$$

We also assume that for every operator op of arity $T \rightarrow U$ there is an operator op_r of arity $T \times U \rightarrow T$. The idea is that $\mathrm{op}_r(\langle L, M \rangle)$ is the reverse-mode derivative of op at L evaluated at M. We write $M.\mathrm{op}_r(L)$ for $\mathrm{op}_r(\langle L, M \rangle)$. For example, for $DProd_2$ we would have:

$$\mathrm{ev}(DProd_2, \langle \langle a, b \rangle, \langle a', b' \rangle \rangle) = aa' + bb'$$

and

$$\mathrm{ev}((DProd_2)_r, \langle \langle \langle a, b \rangle, \langle a', b' \rangle \rangle, c \rangle) = \langle \langle ca', cb' \rangle, \langle ca, cb \rangle \rangle$$

We next define *(value) environments* ρ, *function environments* φ, and *(recursive function) closures* Cl, the last two mutually recursively:

- Value environments are finite functions

$$\rho = \{x_0 \mapsto V_0, \ldots, x_{n-1} \mapsto V_{n-1}\}$$

   from ordinary variables to closed values.
- Every finite function

$$\varphi = \{f_0 \mapsto \mathrm{Cl}_0, \ldots, f_{n-1} \mapsto \mathrm{Cl}_{n-1}\}$$

   from function variables to closures is a function environment.
- If $\mathrm{FV}(M) \subseteq \{x\}$ and $\mathrm{FFV}(M) \backslash \{f\} \subseteq \mathrm{Dom}(\varphi)$ then $\langle \varphi, f, x, T, U, M \rangle$ is a closure, written $\mathbf{clo}_\varphi(f(x : T) : U.\,M)$.

For any V and ρ with $\mathrm{FV}(V) \subseteq \mathrm{Dom}(\rho)$, $\rho(V)$ is the closed value obtained by substituting $\rho(x)$ for all free occurrences of x in V.

Trace terms $C, D, \ldots$, are defined as follows:

$$
\begin{aligned}
C \quad ::= \quad & x \mid r \ (r \in \mathbb{R}) \mid C + D \mid \mathrm{op}(C) \mid \\
& \mathtt{let}\ x{:}T\ =\ C\ \mathtt{in}\ D \mid \\
& * \mid \langle C, D \rangle_{T,U} \mid \mathsf{fst}_{T,U}(C) \mid \mathsf{snd}_{T,U}(C)
\end{aligned}
$$

They are the terms with no conditionals, function definitions or applications, or differentiations.

We will define two *ordinary* evaluation relations, and one *symbolic* one:

- For all φ and ρ we define evaluation relations between terms and closed values and between boolean terms and closed boolean values via rules establishing sequents of the forms:
$$
\varphi \mid \rho \vdash M \Rightarrow V \qquad \varphi \mid \rho \vdash B \Rightarrow V_{\mathsf{bool}}
$$
- For all φ and ρ we define a symbolic evaluation relation between terms and trace terms via rules establishing sequents of the form:
$$
\varphi \mid \rho \vdash M \rightsquigarrow C
$$

Evaluation contexts (boolean evaluation contexts), ranged over by E (resp. E_{bool}), are terms with a unique *hole* [ ] :

$$
\begin{aligned}
E \quad ::= \quad & [\ ] \mid E + N \mid V + E \mid \mathrm{op}(E) \mid \\
& \mathtt{let}\ x{:}T\ =\ E\ \mathtt{in}\ N \mid \\
& \langle E, N \rangle_{T,U} \mid \langle V, E \rangle_{T,U} \mid \mathsf{fst}_{T,U}(E) \mid \mathsf{snd}_{T,U}(E) \mid \\
& \mathtt{if}\ E_{\mathsf{bool}}\ \mathtt{then}\ M\ \mathtt{else}\ N \mid \\
& f(E) \mid \\
& M.\mathrm{rd}_E(x : T.\,N) \mid E.\mathrm{rd}_V(x : T.\,N)
\end{aligned}
$$

$$
E_{\mathsf{bool}} \quad ::= \quad \mathrm{pred}(E)
$$

We write $E[M]$ for the term obtained by replacing the hole [ ] in E by the term M and $E_{bool}[M]$ similarly; a context E is *trivial* if it is [ ]; and FV and FFV are extended to contexts. We have $\mathrm{FV}(E[M]) = \mathrm{FV}(E) \cup \mathrm{FV}(M)$ and $\mathrm{FFV}(E[M]) = \mathrm{FFV}(E) \cup \mathrm{FFV}(M)$ and similarly for boolean contexts.

Redexes, ranged over by R, and *boolean redexes*, ranged over by R_{bool}, are given by:

$$
\begin{aligned}
R \quad ::= \quad & V + W \mid \mathrm{op}(V) \mid \\
& \mathtt{let}\ x{:}T\ =\ V\ \mathtt{in}\ N \mid \\
& \mathsf{fst}_{T,U}(V) \mid \mathsf{snd}_{T,U}(V) \mid \\
& \mathtt{if}\ V_{\mathsf{bool}}\ \mathtt{then}\ M\ \mathtt{else}\ N \mid \\
& \mathtt{letrec}\ f(x{:}T){:}U\ =\ M\ \mathtt{in}\ N \mid f(V) \mid \\
& W.\mathrm{rd}_V(x{:}T.\,N)
\end{aligned}
$$

$$
R_{\mathsf{bool}} \quad ::= \quad \mathrm{pred}(V)
$$

Note that boolean expressions are useful here in that they enable separate conditional and predicate redexes, and so evaluating predicates and making choices are distinct in the operational semantics.

The next lemma is the basis of a division into cases that supports operational semantics using evaluation contexts in the style of Felleisen and Friedman.

LEMMA 3.1 (EVALUATION CONTEXT ANALYSIS).
(1) Every term M, other than a value, has exactly one of the following two forms:
- *$E[R]$ for a unique evaluation context and redex, or*
- *$E[R_{\mathsf{bool}}]$ for a unique, and non-trivial, evaluation context and boolean redex.*

(2) Every boolean term B, other than a boolean value, has exactly one of the following two forms:

- $E_{bool}[R]$ *for a unique, and non-trivial, boolean evaluation context and redex, or*
- $E_{bool}[R_{\mathsf{bool}}]$ *for a unique boolean evaluation context and boolean redex.*

The next lemma is useful to track types when proving theorems about the operational semantics.

LEMMA 3.2 (EVALUATION CONTEXT POLYMORPHISM). *Suppose that* $\Phi \mid \Gamma \vdash E[M]\!:\!T$. *Then, for some type* U *we have* $\Phi \mid \Gamma \vdash M\!:\!U$ *and, whenever* $\Phi \mid \Gamma \vdash N\!:\!U$, *we have* $\Phi \mid \Gamma \vdash E[N]\!:\!T$.

Analogous results hold for typings of any of the forms $\Phi \mid \Gamma \vdash E[B] : T$ *or* $\Phi \mid \Gamma \vdash E_{\mathsf{bool}}[M]$ *or* $\Phi \mid \Gamma \vdash E_{\mathsf{bool}}[B]$.

By the uniqueness of types, the types whose existence is claimed in the above lemma are unique.

$$\varphi \mid \rho \vdash V \Rightarrow \rho(V)$$

$$\frac{\varphi\mid\rho\vdash V\Rightarrow r \quad \varphi\mid\rho\vdash W\Rightarrow s}{\varphi\mid\rho\vdash V+W\Rightarrow t}$$
$$(\text{where } t = r + s)$$

$$\frac{\varphi\mid\rho\vdash V\Rightarrow V'}{\varphi\mid\rho\vdash\mathrm{op}(V)\Rightarrow W}$$
$$(\text{where ev(op}, V') \simeq W)$$

$$\frac{\varphi\mid\rho\vdash V\Rightarrow V' \quad \varphi\mid\rho[V'/x]\vdash N\Rightarrow W}{\varphi\mid\rho\vdash\mathtt{let}\ xT = V\ \mathtt{in}\ N\Rightarrow W}$$

$$\frac{\varphi\mid\rho\vdash V\Rightarrow\langle W_1,W_2\rangle_{T,U}}{\varphi\mid\rho\vdash\mathsf{fst}_{T,U}(V)\Rightarrow W_1}$$

$$\frac{\varphi\mid\rho\vdash V\Rightarrow\langle W_1,W_2\rangle_{T,U}}{\varphi\mid\rho\vdash\mathsf{snd}_{T,U}(V)\Rightarrow W_2}$$

$$\frac{\varphi\mid\rho\vdash M\Rightarrow V}{\varphi\mid\rho\vdash\mathtt{if\ true\ then}\ M\ \mathtt{else}\ N\Rightarrow V}$$

$$\frac{\varphi\mid\rho\vdash N\Rightarrow V}{\varphi\mid\rho\vdash\mathtt{if\ false\ then}\ M\ \mathtt{else}\ N\Rightarrow V}$$

$$\frac{\varphi[\mathbf{clo}_\varphi(f(x\!:\!T)\!:\!U.\,M)/f]\mid\rho\vdash N\Rightarrow V}{\varphi\mid\rho\vdash\mathtt{letrec}\ f(x\!:\!T)\!:\!U = M\ \mathtt{in}\ N\Rightarrow V}$$

$$\frac{\varphi\mid\rho\vdash V\Rightarrow V' \quad \varphi'[\varphi(f)/f]\mid\{x\mapsto V'\}\vdash M\Rightarrow W}{\varphi\mid\rho\vdash f(V)\Rightarrow W}$$
$$(\text{where } \varphi(f) = \mathbf{clo}_{\varphi'}(f(x : T) : U.\,M))$$

$$\frac{\varphi\mid\rho\vdash W.\mathsf{rd}_V(x\!:\!T.\,N)\rightsquigarrow C \quad \varphi\mid\rho\vdash C\Rightarrow X}{\varphi\mid\rho\vdash W.\mathsf{rd}_V(x\!:\!T.\,N)\Rightarrow X}$$

$$\frac{\varphi\mid\rho\vdash V\Rightarrow V'}{\varphi\mid\rho\vdash\mathsf{pred}(V)\Rightarrow W_{\mathsf{bool}}}$$
$$(\text{where bev(pred}, V') \simeq W_{\mathsf{bool}})$$

Fig. 2. Ordinary operational semantics: values and redexes

$$\frac{\varphi\mid\rho\vdash R\Rightarrow V \quad \varphi\mid\rho[V/x]\vdash E[x]\Rightarrow W}{\varphi\mid\rho\vdash E[R]\Rightarrow W} \qquad (E\ \text{nontrivial}, x\notin\mathrm{Dom}(\rho))$$

$$\frac{\varphi\mid\rho\vdash R_{\mathsf{bool}}\Rightarrow V_{\mathsf{bool}} \quad \varphi\mid\rho\vdash E[V_{\mathsf{bool}}]\Rightarrow W}{\varphi\mid\rho\vdash E[R_{\mathsf{bool}}]\Rightarrow W}$$

$$\frac{\varphi\mid\rho\vdash R\Rightarrow V \quad \varphi\mid\rho[V/x]\vdash E_{\mathsf{bool}}[x]\Rightarrow W_{\mathsf{bool}}}{\varphi\mid\rho\vdash E_{\mathsf{bool}}[R]\Rightarrow W_{\mathsf{bool}}} \qquad (x\notin\mathrm{Dom}(\rho))$$

$$\frac{\varphi\mid\rho\vdash R_{\mathsf{bool}}\Rightarrow V_{\mathsf{bool}} \quad \varphi\mid\rho\vdash E_{\mathsf{bool}}[V_{\mathsf{bool}}]\Rightarrow W_{\mathsf{bool}}}{\varphi\mid\rho\vdash E_{\mathsf{bool}}[R_{\mathsf{bool}}]\Rightarrow W_{\mathsf{bool}}}$$

Fig. 3. Ordinary operational semantics: contexts

The rules for ordinary evaluation are given in Figures 2 and 3; those for symbolic evaluation are given in Figures 4 and 5. The definitions are mutually recursive. They make use of the symbolic

$$\varphi \mid \rho \vdash V \leadsto V$$

$$\frac{\varphi|\rho\vdash M \leadsto C}{\varphi|\rho\vdash \text{if true then } M \text{ else } N \leadsto C}$$

$$\varphi \mid \rho \vdash V + W \leadsto V + W$$

$$\frac{\varphi|\rho\vdash N \leadsto C}{\varphi|\rho\vdash \text{if false then } M \text{ else } N \leadsto C}$$

$$\varphi \mid \rho \vdash \text{op}(V) \leadsto \text{op}(V)$$

$$\frac{\varphi[\mathbf{clo}_\varphi(f(x{:}T){:}U.\,M)/f]|\rho\vdash N \leadsto C}{\varphi|\rho\vdash \text{ letrec } f(x{:}T){:}U = M \text{ in } N \leadsto C}$$

$$\frac{\varphi|\rho\vdash V \Rightarrow V' \quad \varphi|\rho[V'/x]\vdash N \leadsto C}{\varphi|\rho\vdash \text{let } x{:}T = V \text{ in } N \leadsto \text{let } x{:}T = V \text{ in } C}$$

$$\frac{\varphi|\rho\vdash V \Rightarrow V' \quad \varphi'[\varphi(f)/f]|\{x \mapsto V'\}\vdash M \leadsto C}{\varphi|\rho\vdash f(V) \leadsto \text{let } x{:}T = V \text{ in } C}$$
$$(\text{where } \varphi(f) = \mathbf{clo}_{\varphi'}(f(x : T) : U.\,M))$$

$$\varphi \mid \rho \vdash \text{fst}_{T,U}(V) \leadsto \text{fst}_{T,U}(V)$$

$$\varphi \mid \rho \vdash \text{snd}_{T,U}(V) \leadsto \text{snd}_{T,U}(V)$$

$$\frac{\varphi|\rho\vdash V \Rightarrow V' \quad \varphi|\rho[V'/x]\vdash N \leadsto C}{\varphi|\rho\vdash W.\text{rd}_V(x{:}T.\,N) \leadsto W.\mathcal{R}_V(x{:}T.\,C)}$$

Fig. 4. Symbolic operational semantics: values and redexes

$$\frac{\varphi|\rho\vdash R \leadsto C \quad \varphi|\rho\vdash C \Rightarrow V \quad \varphi|\rho[V/x]\vdash E[x] \leadsto D}{\varphi|\rho\vdash E[R] \leadsto \text{let } x{:}T_V = C \text{ in } D} \quad (E \text{ nontrivial and } x \notin \text{Dom}(\rho))$$

$$\frac{\varphi|\rho\vdash R_\text{bool} \Rightarrow V_\text{bool} \quad \varphi|\rho\vdash E[V_\text{bool}] \leadsto C}{\varphi|\rho\vdash E[R_\text{bool}] \leadsto C}$$

Fig. 5. Symbolic operational semantics: contexts

differentiation of trace terms: given a trace term C, and values V and W (not necessarily closed), we define a trace term

$$W.\mathcal{R}_V(x{:}T.\,C)$$

intended to denote the reverse-mode derivative of the function $x{:}T \mapsto C$, at V, evaluated at W. A definition is given in Figure 6; in the definition we assume that $x \notin \text{FV}(V, W)$, and, as is common, that all binding variables are different.

PROPOSITION 3.3. *The following typing rule is admissible:*

$$\frac{\Gamma[x{:}T] \vdash C{:}U \quad \Gamma \vdash V{:}T \quad \Gamma \vdash W{:}U}{\Gamma \vdash W.\mathcal{R}_V(x{:}T.\,C){:}T}$$

In large part because of the restrictions on trace terms, their symbolic differentiation is just a systematic, formal application of the chain rule. In our setting, this application requires a fair amount of attention to detail, for instance the use of the type decorations when giving derivatives of pairing and projection terms.

The reader may wish to try the following two evaluation examples with nested differentiation:

$$1.\text{rd}_1(x{:}\text{real}.\,x \times 1.\text{rd}_1(y{:}\text{real}.\,x + y)) \Rightarrow 1$$

and

$$\texttt{letrec } f(x{:}\text{real}){:}\text{real} = 1.\text{rd}_1(y{:}\text{real}.\,x + y) \texttt{ in } 1.\text{rd}_1(x{:}\text{real}.x + f(x)) \Rightarrow 1$$

Examples of this kind can be used to illustrate perturbation confusion in forward differentiation, e.g., [Siskind and Pearlmutter 2005, 2008].

We need some basic results on our evaluation relations. Two are standard: determinacy and type safety, and are used implicitly throughout the rest of the paper. The third connects symbolic

$$W.\mathcal{R}_V(x\!:\!T.y) \qquad = \begin{cases} W & (y = x) \\ 0_T & (y \neq x) \end{cases}$$

$$W.\mathcal{R}_V(x\!:\!T.r) \qquad = \quad 0_T \qquad (r \in \mathbb{R})$$

$$W.\mathcal{R}_V(x\!:\!T.D\!+\!E) \qquad = \quad W.\mathcal{R}_V(x\!:\!T.D)\!+_T W.\mathcal{R}_V(x\!:\!T.E)$$

$$W.\mathcal{R}_V(x\!:\!T.\mathsf{op}(D)) \qquad = \quad \begin{array}{l} \mathtt{let}\ x\!:\!T\ =\ V\ \mathtt{in} \\ \mathtt{let}\ y\!:\!S\ =\ W.\mathsf{op}_r(D)\ \mathtt{in}\ y.\mathcal{R}_V(x\!:\!T.D) \\ \quad (y \notin \mathrm{FV}(V), \mathsf{op}\!:\!S \to U) \end{array}$$

$$W.\mathcal{R}_V(x\!:\!T.\mathtt{let}\ y\!:\!S\ =\ D\ \mathtt{in}\ E) \quad = \quad \begin{array}{l} \mathtt{let}\ x\!:\!T = V\ \mathtt{in} \\ \mathtt{let}\ y\!:\!S\ =\ D\ \mathtt{in} \\ W.\mathcal{R}_V(x\!:\!T.E)\ +_T \\ (\mathtt{let}\ \overline{y}\!:\!S\ =\ W.\mathcal{R}_y(y\!:\!S.E)\ \mathtt{in}\ \overline{y}.\mathcal{R}_V(x\!:\!T.D)) \\ \quad (y \notin \mathrm{FV}(W), y, \overline{y} \notin \mathrm{FV}(V, D)) \end{array}$$

$$W.\mathcal{R}_V(x\!:\!T.\ast) \qquad = \quad 0_T$$

$$W.\mathcal{R}_V(x\!:\!T.\ \langle D, E\rangle_{U,S}) \qquad = \quad \begin{array}{l} \mathtt{let}\ y\!:\!U, z\!:\!S\ =\ W\ \mathtt{in} \\ y.\mathcal{R}_V(x\!:\!T.\ D)\ +_T\ z.\mathcal{R}_V(x\!:\!T.\ E) \\ \quad (y, z \notin \mathrm{FV}(V, D, E)) \end{array}$$

$$W.\mathcal{R}_V(x\!:\!T.\mathsf{fst}_{U,S}(D)) \qquad = \quad \begin{array}{l} \mathtt{let}\ x\!:\!T\ =\ V\ \mathtt{in} \\ \mathtt{let}\ y\!:\!U \times S\ =\ D\ \mathtt{in}\ \langle W, 0_S\rangle.\mathcal{R}_V(x\!:\!T.D) \\ \quad (y \notin \mathrm{FV}(V, W, D)) \end{array}$$

$$W.\mathcal{R}_V(x\!:\!T.\mathsf{snd}_{U,S}(D)) \qquad = \quad \begin{array}{l} \mathtt{let}\ x\!:\!T\ =\ V\ \mathtt{in} \\ \mathtt{let}\ y\!:\!U \times S\ =\ D\ \mathtt{in}\ \langle 0_U, W\rangle.\mathcal{R}_V(x\!:\!T.D) \\ \quad (y \notin \mathrm{FV}(V, W, D)) \end{array}$$

Fig. 6. Definition of $W.\mathcal{R}_V(x\!:\!T.C)$

and ordinary evaluation: one can interpolate symbolic evaluation within ordinary evaluation. It is principally helpful to reduce the completeness part of symbolic evaluation to the completeness of ordinary evaluation (see Theorem 6.7).

PROPOSITION 3.4 (DETERMINACY OF EVALUATION). *The following hold:*

(1) For any φ, ρ, and M, there is at most one value V s.t. $\varphi \mid \rho \vdash M \Rightarrow V$.
(2) For any φ, ρ, and M, there is at most one trace term C s.t. $\varphi \mid \rho \vdash M \rightsquigarrow C$.

The following interpolation proposition establishes a certain consistency between the ordinary and symbolic evaluation relations.

PROPOSITION 3.5 (OPERATIONAL INTERPOLATION). *For all φ, ρ, and closed values V, the following are equivalent:*

(1) $\varphi \mid \rho \vdash M \Rightarrow V$,
(2) $\varphi \mid \rho \vdash M \rightsquigarrow C$ and $\varphi \mid \rho \vdash C \Rightarrow V$, for some C.

For a type safety theorem, we need typing judgments $\rho : \Gamma$, $\varphi : \Phi$, and $\mathrm{Cl} : T \to U$ for environments, function environments, and closures (implicitly extending the notion of type). These are defined inductively by the following rules:

$$\frac{\vdash V_i : T_i \quad (i = 0, n-1)}{\vdash \{x_0 \mapsto V_0, \ldots, x_{n-1} \mapsto V_{n-1}\} \, : \, x_0 : T_0, \ldots, x_{n-1} : T_{n-1}}$$

$$\frac{\vdash \mathrm{Cl}_i : T_i \to U_i \quad (i = 0, n-1)}{\vdash \{f_0 \mapsto \mathrm{Cl}_0, \ldots, f_{n-1} \mapsto \mathrm{Cl}_{n-1}\} \, : \, f_0 : T_0 \to U_0, \ldots, f_{n-1} : T_{n-1} \to U_{n-1}}$$

$$\frac{\vdash \varphi : \Phi \quad \Phi[f : T \to U] \mid x : T \vdash M : U}{\vdash \mathbf{clo}_\varphi(f(x : T) : U. M) : T \to U}$$

Note that a closure $\mathbf{clo}_\varphi(f(x : T) : U. M)$ can only have type $T \to U$. So in the third rule Φ is determined up to the ordering of its function type declarations. Whether the conclusion of the rule follows does not depend on the choice of this ordering. We write Φ_φ for the choice of Φ with declarations ordered using the standard function variable ordering.

PROPOSITION 3.6 (TYPE SAFETY). *Suppose* $\Phi \mid \Gamma \vdash M : T$, $\vdash \varphi : \Phi$ *and* $\vdash \rho : \Gamma$. *Then we have:*

$$\varphi \mid \rho \vdash M \Rightarrow V \implies \vdash V : T$$

and

$$\varphi \mid \rho \vdash M \rightsquigarrow C \implies \Gamma \vdash C : T$$

4 MATHEMATICAL PRELIMINARIES

We now turn to the mathematical facts needed for the denotational semantics of our language. These concern the two modes of differentiation and their interaction with domain theory. We follow [Abramsky and Jung 1994] for domain theory, but write dcppo for pointed dcpo, and say a partial order is *coherent* iff every compatible subset has a lub. (A subset is *compatible* if any two of its elements have an upper bound.) Every coherent partial order is a dcppo.

The collection of partial functions $f : X \rightharpoonup Y$ with open domain between two topological spaces forms a partial order under graph inclusion:

$$f \leq g \iff f \subseteq g$$

equivalently, using the Kleene order[4]:

$$f \leq g \iff \forall \mathbf{x} \in X. \, f\mathbf{x} \leq g\mathbf{x}$$

This partial order is a coherent dcppo with $\perp$ the everywhere undefined function and compatible sups given by unions. A partial function $f : X \rightharpoonup Y$ is *continuous* if $f^{-1}(B)$ is open whenever B is; the subcollection of continuous partial functions forms a coherent subdcppo. This holds as, for any open set $B \subseteq Y$ and compatible collection of partial functions $f_i \, (i \in I) : X \rightharpoonup Y$, we have:

$$\left(\bigvee_{i \in I} f_i\right)^{-1}(B) = \bigcup_{i \in I} f_i^{-1}(B)$$

We write $C[X, Y]$ for the dcppo of partial continuous functions from X to Y.

It is convenient to use a variation on cartesian product when working with powers of $\mathbb{R}$. We set:

$$\mathbb{R}^m \mathbin{\dot{\times}} \mathbb{R}^n =_{\mathrm{def}} \mathbb{R}^{m+n} \quad (m, n \geq 0)$$

---

[4]We write $e \leq e'$ for two mathematical expressions e and e' to mean that if e is defined so is e', and they are then equal.

This version of product is associative. Vector concatenation then serves as tupling; however, for clarity, we may use the usual notation $(\mathbf{x}_0, \ldots, \mathbf{x}_{k-1})$ instead of $\mathbf{x}_0 \ldots \mathbf{x}_{k-1}$. There are evident definitions of the projections $\pi_i^{m_0,\ldots,m_{k-1}} : \mathbb{R}^{m_0} \,\dot{\times}\, \ldots \,\dot{\times}\, \mathbb{R}^{m_{k-1}} \to \mathbb{R}^{m_i}$, and of the tupling

$$\langle f_0, \ldots, f_{k-1} \rangle : \mathbb{R}^n \rightharpoonup \mathbb{R}^{m_0} \,\dot{\times}\, \ldots \,\dot{\times}\, \mathbb{R}^{m_{k-1}}$$

of $f_i : \mathbb{R}^n \rightharpoonup \mathbb{R}^{m_i}$. We may ignore the superscripts on the projections when they can be understood from the context.

4.1 Continuity, Differentiability, and Smoothness

Standard multivariate analysis of vector-valued real functions from $\mathbb{R}^n$ to $\mathbb{R}^m$ (with $n, m > 0$) considers functions f defined on an open domain of $\mathbb{R}^n$, see, e.g., [Trench 2003]. These are precisely the partial functions:

$$f : \mathbb{R}^n \rightharpoonup \mathbb{R}^m \quad (n, m > 0)$$

with open domain.

When $m = 1$, such a function f has partial derivatives

$$\partial_j(f) : \mathbb{R}^n \rightharpoonup \mathbb{R} \qquad (j = 0, n-1)$$

where

$$\partial_j(f)(x_0, \ldots, x_{n-1}) \simeq_{\text{def}} \frac{\partial f}{\partial x_j}$$

viewing f as a function of $x_0, \ldots, x_{n-1}$. Taken together, these partial derivatives form its *gradient*

$$\nabla(f) : \mathbb{R}^n \rightharpoonup \mathbb{R}^n$$

where

$$\nabla(f)(\mathbf{x}) \simeq \langle \partial_0(f)(\mathbf{x}), \ldots, \partial_{n-1}(f)(\mathbf{x}) \rangle \tag{1}$$

We write $\nabla_{\mathbf{x}}(f)$ for $\nabla(f)(\mathbf{x})$.

We say f is *continuously differentiable* if all the $\partial_j(f)$ are continuous with domain that of f, equivalently if $\nabla(f)$ is continuous with domain that of f. As an example, removing 0 from the domain of definition of the non-differentiable ReLU function $f(x) = \max(x, 0)$, we obtain a continuously differentiable partial function with domain $\mathbb{R}\backslash 0$; its derivative also has domain $\mathbb{R}\backslash 0$, with value 0, if $x < 0$, and 1, if $x > 0$.

We now turn to the general case where $f : \mathbb{R}^n \rightharpoonup \mathbb{R}^m$. The *Jacobian* $J_{\mathbf{x}}(f)$ of f at $\mathbf{x} \in \mathbb{R}^n$ is the m by n matrix:

$$J_{\mathbf{x}}(f)_{i,j} \simeq \partial_j(\pi_i \circ f)(\mathbf{x}) \tag{2}$$

where the matrix is undefined if any of the $\partial_j(\pi_i \circ f)(\mathbf{x})$ are. These Jacobians form a partial function:

$$J(f) : \mathbb{R}^n \rightharpoonup \text{Mat}(m, n)$$

where $\text{Mat}(m, n)$ is the collection of m by n matrices. Viewing $\text{Mat}(m, n)$ as $\mathbb{R}^{m \times n}$, we say f is *continuously differentiable* if $J(f)$ is continuous and has the same domain as f (equivalently if each component $\pi_i \circ f$ of f is continuously differentiable).

The *differential* [5]

$$d(f) : \mathbb{R}^n \,\dot{\times}\, \mathbb{R}^n \rightharpoonup \mathbb{R}^m$$

of f is defined by:

$$d(f)(\mathbf{x}, \mathbf{y}) \simeq J_{\mathbf{x}}(f) \cdot \mathbf{y} \tag{3}$$

and f is continuously differentiable iff $d(f)$ has domain $\text{Dom}(f) \times \mathbb{R}^n$ and is continuous there.

---

[5]Differentials are discussed in [Trench 2003] see: p325 for differentials of functions of several variables; p348 for higher-order differentials; p381 for differentials of vector-valued functions of several variables; and p388 for the chain rule in differential terms.

We write $d_{\mathbf{x}}(f)$ for the partial function $d(f)(\mathbf{x}, -)$; it is either $\perp$ or everywhere defined and linear, the latter occurring precisely when $J_{\mathbf{x}}(f)$ is defined. If f is linear then $d_{\mathbf{x}}f = f$, for all $\mathbf{x} \in \mathbb{R}^n$.

In the automatic differentiation literature, $d_{\mathbf{x}}(f)$ is called the *forward-mode* derivative of f at $\mathbf{x}$. For the *reverse-mode* derivative we define:

$$d^r(f):\mathbb{R}^n \dot{\times} \mathbb{R}^m \rightharpoonup \mathbb{R}^n$$

by:

$$d^r(f)(\mathbf{x}, \mathbf{y}) \simeq J_{\mathbf{x}}(f)^t \cdot \mathbf{y} \qquad (4)$$

and write $d_{\mathbf{x}}^r(f)$ for the partial function $d^r(f)(\mathbf{x}, -)$; f is continuously differentiable iff $d^r(f)$ has domain $\mathrm{Dom}(f) \times \mathbb{R}^m$ and is continuous there.

In terms of the differentials, the two modes are related by:

$$d_{\mathbf{x}}^r(f) = d_{\mathbf{x}}(f)^\dagger \qquad (5)$$

(setting $\perp^\dagger = \perp$). So, if f is linear, then $d_{\mathbf{x}}^r(f) = d_{\mathbf{x}}(f)^\dagger = f^\dagger$.

The semantic content of the definition of forward-mode from reverse-mode given in Section 2 is the following equality:

$$d_{\mathbf{x}}f = d_0^R(d_{\mathbf{x}}^R f) \qquad (6)$$

This holds as: $d_0^R(d_{\mathbf{x}}^R f) = (d_{\mathbf{x}}^R f)^\dagger = ((d_{\mathbf{x}}f)^\dagger)^\dagger = d_{\mathbf{x}}f$ (the first equality holds as $d_{\mathbf{x}}^R f$ is linear if $\neq \perp$).

The continuously differentiable functions are closed under composition. If $h:\mathbb{R}^n \rightharpoonup \mathbb{R}^l$ is the composition of two such functions $f:\mathbb{R}^n \rightharpoonup \mathbb{R}^m$ and $g:\mathbb{R}^m \rightharpoonup \mathbb{R}^l$, then the chain rule expresses the derivative of h in terms of those of f and g. In terms of Jacobians, the chain rule is:

$$J_{\mathbf{x}}(h) \quad \simeq \quad J_{f(\mathbf{x})}(g) \cdot J_{\mathbf{x}}(f) \quad (\mathbf{x} \in \mathrm{Dom}(h))$$

(Note that $\mathbf{x} \in \mathrm{Dom}(h)$ iff $\mathbf{x} \in \mathrm{Dom}(f)$ and $f(\mathbf{x}) \in \mathrm{Dom}(g)$.) In terms of forward-mode derivatives the chain rule is:

$$d_{\mathbf{x}}(h) \quad = \quad d_{f(\mathbf{x})}(g) \circ d_{\mathbf{x}}(f) \quad (\mathbf{x} \in \mathrm{Dom}(h)) \qquad (7)$$

and in terms of reverse-mode derivatives it is:

$$d_{\mathbf{x}}^r(h) \quad = \quad d_{\mathbf{x}}^r(f) \circ d_{f(\mathbf{x})}^r(g) \quad (\mathbf{x} \in \mathrm{Dom}(h)) \qquad (8)$$

Derivatives with respect to two variables can be reduced to derivatives in each separately. Specifically, suppose $f:\mathbb{R}^{n+n'} \rightharpoonup \mathbb{R}^m$. Then, for $\mathbf{x} \in \mathbb{R}^n$ and $\mathbf{y} \in \mathbb{R}^{n'}$, we have:

$$d_{\langle \mathbf{x}, \mathbf{y} \rangle}(f)(\mathbf{u}, \mathbf{v}) \simeq d_{\mathbf{x}}(f(-, \mathbf{y}))(\mathbf{u}) + d_{\mathbf{y}}(f(\mathbf{x}, -))(\mathbf{v}) \quad (\mathbf{u} \in \mathbb{R}^n, \mathbf{v} \in \mathbb{R}^{n'}) \qquad (9)$$

and

$$d_{\langle \mathbf{x}, \mathbf{y} \rangle}^R(f) = \langle d_{\mathbf{x}}^R(f(-, \mathbf{y})), d_{\mathbf{y}}^R(f(\mathbf{x}, -)) \rangle \qquad (10)$$

These equations are useful for dealing with fan-in.

Our programming language has all finite product types of the reals. We will therefore need to work with partial functions with open domain $f:\mathbb{R}^n \rightharpoonup \mathbb{R}^m$ where n or m is zero. To this end we regard $\mathbb{R}^0$ as having as sole element the empty vector, the trivial topology, and the trivial vector space structure. Every such total function is linear, and this determines its adjoint.

We take such a function $f:\mathbb{R}^n \rightharpoonup \mathbb{R}^m$, where n or m is zero, to be continuously differentiable if it is continuous, and we define $d(f):\mathbb{R}^n \dot{\times} \mathbb{R}^n \rightharpoonup \mathbb{R}^m$ and $d^r(f):\mathbb{R}^n \dot{\times} \mathbb{R}^m \rightharpoonup \mathbb{R}^n$ by:

$$d(f)(\mathbf{x}, \mathbf{y}) \simeq \begin{cases} 0 & (f(\mathbf{x}) \downarrow) \\ \uparrow & (\text{otherwise}) \end{cases} \qquad d^r(f)(\mathbf{x}, \mathbf{y}) \simeq \begin{cases} 0 & (f(\mathbf{x}) \downarrow) \\ \uparrow & (\text{otherwise}) \end{cases}$$

The derivative $d(f)$ has domain $\mathrm{Dom}(f) \times \mathbb{R}^n$ and is continuous (and so continuously differentiable) iff f is, and a similar remark applies to $d^r(f)$. We understand $d_{\mathbf{x}}(f)$ and $d_{\mathbf{x}}^r(f)$ similarly to before. In case f is linear (i.e., total) we have $d_{\mathbf{x}}(f) = f$ and $d_{\mathbf{x}}^r(f) = f^\dagger$ as before and equation (5) relating

the two modes continues to hold, as does equation (6) and also equations (10) and (9) concerning derivatives with respect to two variables. In particular for $t:\mathbb{R}^n \to \mathbb{R}^0$ we have:

$$(d_{\mathbf{x}}t)\mathbf{x}' \quad = \quad * \qquad (d_{\mathbf{x}}^r t)* = \mathbf{0}$$

Regarding compositions we note a useful fact:

FACT 1. *If* $h:\mathbb{R}^n \rightharpoonup \mathbb{R}^l$ $(l, n \geq 0)$ *is constant on its domain, then it is continuously differentiable iff it is continuous and then, for any* $\mathbf{x}$ *in its domain,* $d_{\mathbf{x}}(h)$ *is the constantly* 0 *function, as is* $d_{\mathbf{x}}^r(h)$.

It follows that the continuously differentiable functions, if taken in our wider sense, remain closed under composition and pointwise addition, and that the chain rule for forward derivatives continues to hold, as does that for reverse derivatives. From now on whenever we consider partial functions from an $\mathbb{R}^n$ to an $\mathbb{R}^m$, we include the cases where n or m is 0.

The projections $\pi_i^{m_0,\dots,m_{k-1}}:\mathbb{R}^{m_0} \dot{\times} \dots \dot{\times} \mathbb{R}^{m_{k-1}} \to \mathbb{R}^{m_i}$ are total linear functions, so we have:

$$
\begin{aligned}
d_{\mathbf{x}}\pi_i^{m_0,\dots,m_{k-1}} &= \pi_i^{m_0,\dots,m_{k-1}} \\
(d_{\mathbf{x}}^r\pi_i^{m_0,\dots,m_{k-1}})\mathbf{y} &= (0,\dots,0,\mathbf{y},0,\dots,0)
\end{aligned}
$$

Regarding the tupling $\langle f_0,\dots,f_{k-1}\rangle:\mathbb{R}^n \rightharpoonup \mathbb{R}^{m_0} \dot{\times} \dots \dot{\times} \mathbb{R}^{m_{k-1}}$ of $f_i:\mathbb{R}^n \rightharpoonup \mathbb{R}^{m_i}$ we have:

$$
\begin{aligned}
d_{\mathbf{x}}\langle f_0,\dots,f_{k-1}\rangle &= \langle d_{\mathbf{x}}f_0,\dots,d_{\mathbf{x}}f_{k-1}\rangle \\
(d_{\mathbf{x}}^r\langle f_0,\dots,f_{k-1}\rangle)(\mathbf{y}_0,\dots,\mathbf{y}_{k-1}) &\simeq (d_{\mathbf{x}}^r f_0)\mathbf{y}_0 + \dots + (d_{\mathbf{x}}^r f_{k-1})\mathbf{y}_{k-1}
\end{aligned}
$$

For the semantics of our language we work with infinitely differentiable functions, i.e., smooth ones. First we define smoothness classes C^k. We say that a partial function $f:\mathbb{R}^n \rightharpoonup \mathbb{R}^m$ is C^0 if it is continuous, and, inductively, is C^{k+1} if $d(f)$ has domain $\mathrm{Dom}(f) \times \mathbb{R}^n$ and is C^k. This defines a decreasing sequence of classes of functions, and we say that f is *smooth* or C^∞ if it is C^k for all k. The C^1 functions are precisely the continuously differentiable ones. Using the chain rule for differentials one shows that the C^k functions, and so too the smooth ones, are closed under composition. The projections are smooth, as are all linear functions and the C^k functions, and so too the smooth ones, are closed under tupling.

4.2 Cppos of Differentiable Functions

The subgraph partial order on partial functions between powers of $\mathbb{R}$ interacts well with the differential structure:

PROPOSITION 4.1.

(1) For any $f \leq g:\mathbb{R}^n \rightharpoonup \mathbb{R}^m$ with open domain we have:

$$df = (dg) \upharpoonright (\mathrm{Dom}(f) \times \mathbb{R}^n) \qquad d^r f = (d^r g) \upharpoonright (\mathrm{Dom}(f) \times \mathbb{R}^m)$$

(2) For any compatible family of functions with open domain $f_i:\mathbb{R}^n \rightharpoonup \mathbb{R}^m$ $(i \in I)$, we have:

$$d \bigvee_{i \in I} f_i = \bigvee_{i \in I} df_i \qquad d^r \bigvee_{i \in I} f_i = \bigvee_{i \in I} d^r f_i$$

PROOF. For the first part, if m or n is 0, the conclusion is immediate using Fact 1. Otherwise, as f and g agree on an open set including x we have:

$$\mathrm{J}_{\mathbf{x}}(f) \simeq \mathrm{J}_{\mathbf{x}}(g)$$

and the conclusion follows. For the second part, set $f = \bigvee f_i$. For the forward-mode derivative, using the first part we calculate:

$$
\begin{aligned}
df &= df \upharpoonright (\mathrm{Dom}(f) \times \mathbb{R}^n) &= df \upharpoonright \bigcup_{i \in I}(\mathrm{Dom}(f_i) \times \mathbb{R}^n) \\
&= \bigvee_{i \in I} df \upharpoonright (\mathrm{Dom}(f_i) \times \mathbb{R}^n) &= \bigvee_{i \in I} df_i
\end{aligned}
$$

The proof for the reverse-mode derivative is similar.

$\square$

PROPOSITION 4.2.
(1) Let $f \leq g : \mathbb{R}^n \rightharpoonup \mathbb{R}^m$ be partial functions with open domain. Then f is smooth if g is.
(2) Let $f_i : \mathbb{R}^n \rightharpoonup \mathbb{R}^m$ be a compatible family of partial functions with open domain, and with sup f. Then f is smooth if all the f_i are.

PROOF. For the first part, we prove by induction that if g is C^k then so is f. For $k = 0$ we note that for any open $V \subseteq \mathbb{R}^m$, $f^{-1}(V) = g^{-1}(V) \cap \mathrm{Dom}(f)$. For $k + 1$, as g is C^{k+1}, $\mathrm{Dom}(\mathrm{d}g) = \mathrm{Dom}(g) \times \mathbb{R}^n$ and $\mathrm{d}g$ is C^k. From part (1) of Proposition 4.1 we have $\mathrm{d}f \leq \mathrm{d}g$ and $\mathrm{Dom}(\mathrm{d}f) = \mathrm{Dom}(f) \times \mathbb{R}^n$. So $\mathrm{d}f$ and $\mathrm{d}g$ have open domain, $\mathrm{d}f \leq \mathrm{d}g$, and $\mathrm{d}g$ is C^k. It follows from the induction hypothesis that $\mathrm{d}f$ is C^k. So f is C^{k+1}, as required.

For the second part we prove, by induction on k that if all the f_i are C^k, then so is f. For $k = 0$ this is clear. For $k + 1$, we have, for all i, that $\mathrm{Dom}(\mathrm{d}f_i) = \mathrm{Dom}(f_i) \times \mathbb{R}^n$ and f_i is C^k. From part (2) of Proposition 4.1 we have $\mathrm{d}f = \bigvee_i \mathrm{d}f_i$. So, first,

$$
\begin{array}{rclcl}
\mathrm{Dom}(\mathrm{d}f) & = & \mathrm{Dom}(\bigvee_i \mathrm{d}f_i) & = & \bigcup_i \mathrm{Dom}(\mathrm{d}f_i) \\
& = & \bigcup_i \mathrm{Dom}(f_i) \times \mathbb{R}^n & = & \mathrm{Dom}(f) \times \mathbb{R}^n
\end{array}
$$

and second, also using the induction hypothesis, we have that $\mathrm{d}f$ is C^k. So f is C^{k+1}, as required.

$\square$

We write $\mathcal{S}[\mathbb{R}^n, \mathbb{R}^m]$ for the coherent dcppo of smooth partial functions between $\mathbb{R}^n$ and $\mathbb{R}^m$.

4.3 Conditionals and Recursion

Differentiation and conditionals interact well. The *conditional combinator* $\mathrm{Cond}_{n,m}$ is defined for $p : R^n \rightharpoonup \mathbb{T}$ and $f, g : R^n \rightharpoonup \mathbb{R}^m$ by:

$$
\mathrm{Cond}_{n,m}(p, f, g)(\mathbf{x}) \simeq \left\{
\begin{array}{ll}
f(\mathbf{x}) & (p(\mathbf{x}) = t\!t) \\
g(\mathbf{x}) & (p(\mathbf{x}) = f\!\!f) \\
\uparrow & (\text{otherwise})
\end{array}
\right.
$$

where $\mathbb{T} = \{t\!t, f\!\!f\}$. The conditional combinator is continuous. For differentiability, with $\mathbb{T}$ a discrete topological space, we have:

PROPOSITION 4.3. *Suppose p is continuous (equivalently: both $p^{-1}(t\!t)$ and $p^{-1}(f\!\!f)$ are open). Then:*

$$
\mathrm{d}(\mathrm{Cond}_{n,m}(p, f, g)) \quad = \quad \mathrm{Cond}_{(n+n),m}(p \circ \pi_1, \mathrm{d}f, \mathrm{d}g)
$$

and

$$
\mathrm{d}^r(\mathrm{Cond}_{n,m}(p, f, g)) \quad = \quad \mathrm{Cond}_{(n+m),n}(p \circ \pi_1, \mathrm{d}^r f, \mathrm{d}^r g)
$$

Further, if f, g are smooth so is $\mathrm{Cond}_{n,m}(p, f, g)$.

PROOF. Assume that p is continuous. Set $h = \mathrm{Cond}_{n,m}(p, f, g)$. The domain of h is open, indeed $\mathrm{Dom}(h) = (p^{-1}(t\!t) \cap \mathrm{Dom}(f)) \cup (p^{-1}(f\!\!f) \cap \mathrm{Dom}(g))$ so $\mathrm{d}h$ is defined.

To prove the equality, choose $\mathbf{x} \in \mathbb{R}^n$. There are three cases. First, if $p(\mathbf{x}) \uparrow$ then $h(\mathbf{x}) \uparrow$ and so $(\mathrm{d}h)(\mathbf{x}) \uparrow$; the equality therefore holds at $\mathbf{x}$. Second if $p(\mathbf{x}) = t\!t$ then, as $h \upharpoonright p^{-1}(t\!t) = f \upharpoonright p^{-1}(t\!t)$ and $p^{-1}(t\!t)$ is open, we see that, by part (1) of Proposition 4.1, $\mathrm{d}_{\mathbf{x}} h = \mathrm{d}_{\mathbf{x}} h \upharpoonright p^{-1}(t\!t) = \mathrm{d}_{\mathbf{x}} f \upharpoonright p^{-1}(t\!t) = \mathrm{d}_{\mathbf{x}} f$, and so the equality again holds at $\mathbf{x}$. The third case is similar to the second.

Suppose further that f, g are smooth. We have $h \upharpoonright p^{-1}(t\!t) \leq f$, and so, by part (1) of Proposition 4.2, $h \upharpoonright p^{-1}(t\!t)$ is smooth as f is and $h \upharpoonright p^{-1}(t\!t)$ has open domain. Similarly $h \upharpoonright p^{-1}(f\!\!f)$ is smooth. But then, by part (2) of Proposition 4.2, h is smooth as $h = h \upharpoonright p^{-1}(\text{true}) \vee h \upharpoonright p^{-1}(f\!\!f)$. $\square$

In less formal terms than the proof, equality holds because if, say, the condition p holds at $\mathbf{x} \in \mathbb{R}^n$, it holds in a neighborhood O of $\mathbf{x}$. So f and the conditional are equal throughout O, and therefore have the same derivative there. The equality justifies the approaches to the differentiation of conditionals described in the Introduction.

Differentiation and recursion also interact well. For any continuous $f : P \times Q \to Q$ (P a dcpo, Q a dcppo) we write $\mu y : Q. f(x, y)$ for the least fixed-point (l.f.p.) of $f(x, -)$. It is the sup of the *iterates* $\mu_n y : Q. f(x, y)$, defined inductively by:

$$\mu_{n+1} y : Q. f(x, y) \quad = \quad f(x, \mu_n y : Q. f(x, y))$$

starting from $\perp_Q$. As functions of P, the l.f.p. and the iterates are continuous. When $f : Q \to Q$, we write $\mu y : Q. f(y)$, etc.

PROPOSITION 4.4.

(1) Set $Q = \mathcal{S}[\mathbb{R}^m, \mathbb{R}^l]$ and $R = \mathcal{S}[\mathbb{R}^m \dot{\times} \mathbb{R}^m, \mathbb{R}^l]$. Then if $F : \mathbb{R}^n \times Q \to Q$ and $G : \mathbb{R}^n \times Q \times R \to R$ are such that

$$dF(x, f) = G(x, f, df) \quad (x \in \mathbb{R}^n, f \in Q)$$

we have:

$$d(\mu f : Q. F(x, f)) = \mu f' : R. \, G(x, \mu f : Q. F(x, f), f')$$

(2) Set $Q = \mathcal{S}[\mathbb{R}^m, \mathbb{R}^l]$ and $R = \mathcal{S}[\mathbb{R}^m \dot{\times} \mathbb{R}^l, \mathbb{R}^m]$. Then if $F : \mathbb{R}^n \times Q \to Q$ and $G : \mathbb{R}^n \times Q \times R \to R$ are such that

$$d^r F(x, f) = G(x, f, d^r f) \quad (x \in \mathbb{R}^n, f \in Q)$$

we have:

$$d^r(\mu f : Q. F(x, f)) = \mu f' : R. \, G(x, \mu f : Q. F(x, f), f')$$

PROOF. We only consider the forward-mode case as the reverse-mode case is similar. In one direction we prove by induction on n that

$$d(\mu_n f : Q. F(x, f)) \leq \mu f' : R. \, G(x, \mu f : Q. F(x, f), f')$$

This is evident for $n = 0$. For $n + 1$ we calculate (missing out types):

$$
\begin{aligned}
d(\mu_{n+1} f. F(x, f)) \quad &= \quad d(F(x, \mu_n f. F(x, f))) \\
&= \quad G(x, \mu_n f. F(x, f), d\mu_n f. F(x, f)) \\
&\leq \quad G(x, \mu f. F(x, f), \mu f'. \, G(x, \mu f. F(x, f), f')) \\
&= \quad \mu f'. \, G(x, \mu f. F(x, f), f')
\end{aligned}
$$

In the other direction we prove by induction on n that

$$\mu_n f' : R. \, G(x, \mu f : Q. F(x, f), f') \leq d(\mu f : Q. F(x, f))$$

This is evident for $n = 0$. For $n + 1$ we calculate:

$$
\begin{aligned}
\mu_{n+1} f'. \, G(x, \mu f. F(x, f), f') \quad &= \quad G(x, \mu f. F(x, f), \mu_n f'. \, G(x, \mu f. F(x, f), f')) \\
&\leq \quad G(x, \mu f. F(x, f), d\mu f. F(x, f)) \\
&= \quad dF(x, \mu f. F(x, f)) \\
&= \quad d(\mu f. F(x, f))
\end{aligned}
$$

The conclusion then follows using the continuity of d (part (2) of Proposition 4.1).

□

Although we do not do so here, this proposition can be used to justify code transformations of recursive function definitions.

5 DENOTATIONAL SEMANTICS

We begin with a denotational semantics of types as powers of the reals:

$$
\begin{aligned}
[\![\texttt{real}]\!] &= \mathbb{R} \\
[\![\texttt{unit}]\!] &= \mathbb{R}^0 \\
[\![T \times U]\!] &= [\![T]\!] \,\dot\times\, [\![U]\!]
\end{aligned}
$$

Note that $[\![T]\!] = \mathbb{R}^{|T|}$, where, $|\texttt{real}| = 1$, $|\texttt{unit}| = 0$ and, recursively, $|T \times U| = |T| + |U|$. Then, for the semantics of environments $\Gamma = x_0 : T_0, \ldots, x_{n-1} : T_{n-1}$ we set

$$
[\![\Gamma]\!] = [\![T_0]\!] \,\dot\times\, \ldots \,\dot\times\, [\![T_{n-1}]\!]
$$

and for that of function environments $\Phi = f_0 : T_0 \to U_n, \ldots, f_{n-1} : T_{n-1} \to U_{n-1}$ we set

$$
[\![\Phi]\!] = \mathcal{S}[[\![T_0]\!], [\![U_0]\!]] \times \ldots \times \mathcal{S}[[\![T_{n-1}]\!], [\![U_{n-1}]\!]]
$$

We use γ, δ to range over $[\![\Gamma]\!]$ and ϕ over $[\![\Phi]\!]$. The vectors $\gamma \in [\![\Gamma]\!]$ correspond to semantic environment functions ρ on $\mathrm{Dom}(\Gamma)$ with $\rho(x_i) = \pi_i(\gamma)$, and this correspondence is 1-1. We take advantage of it to write $\gamma[a/x]$, when $a \in [\![T]\!]$, for an element of $[\![\Gamma[x : T]]\!]$ (assuming Γ and T available from the context). We understand $\phi[\alpha/f]$ similarly, for $\phi \in [\![\Phi]\!]$ and $\alpha \in \mathcal{S}[[\![T]\!], [\![U]\!]]$.

The denotational semantics of a term $\Phi \mid \Gamma \vdash M : T$ will be a continuous function:

$$
[\![\Phi]\!] \xrightarrow{[\![\Phi \mid \Gamma \vdash M : T]\!]} \mathcal{S}[[\![\Gamma]\!], [\![T]\!]]
$$

and that of a boolean term $\Phi \mid \Gamma \vdash B$ will be continuous functions:

$$
[\![\Phi]\!] \xrightarrow{[\![\Phi \mid \Gamma \vdash B]\!]} C[[\![\Gamma]\!], \mathbb{T}]
$$

When the environments and types are understood from the context, we may just write $[\![M]\!]$ or $[\![B]\!]$.

For the semantics of operation and predicate symbols, for every $\mathrm{op} : T \to U$ we assume available a smooth function $[\![\mathrm{op}]\!] : [\![T]\!] \to [\![U]\!]$ and for every $\mathrm{pred} : T$ a continuous function $[\![\mathrm{pred}]\!] : [\![T]\!] \to \mathbb{T}$, such that, for every closed value $V : T$:

$$
[\![\mathrm{op}]\!]([\![V]\!]) \simeq [\![\mathrm{ev}(\mathrm{op}, V)]\!] \tag{11}
$$

and, for every closed value $V : T$:

$$
[\![\mathrm{pred}]\!]([\![V]\!]) \simeq [\![\mathrm{bev}(\mathrm{pred}, V)]\!] \tag{12}
$$

The denotational semantics is given in Figure 7. Note that it uses the no-free-variable assumption in the clause for recursive functions. Apart from the semantics of reverse differentiation, which uses the reverse-mode derivative d^r, it is quite standard. However, the facts that the denotations of terms carry smooth functions to smooth functions, and that the denotations of boolean terms carry smooth functions to continuous ones, use the mathematics developed in the previous section, particularly: the chain rule, e.g., for function application and let constructs; the preservation of smooth functions by the conditional combinator; the remarks on products; and, for recursive function definitions, the fact that the lub of an increasing sequence of smooth functions is smooth.

If a term M contains no function variables (or variables), $[\![M]\!]\phi\gamma$ is independent of ϕ (and γ), and we write $[\![M]\!]\gamma$ (resp., $[\![M]\!]$) for it. Trace terms C have no function variables, and closed values V have no function variables or variables.

Using the semantics of terms, we can define the denotational semantics of value environments, function environments, and closures, the latter two by a mutual structural induction.

- For every $\rho : \Gamma$, where $\Gamma = x_0 : T_0, \ldots, x_{n-1} : T_{n-1}$, we define $[\![\rho : \Gamma]\!] \in [\![\Gamma]\!]$ by:

$$
[\![\rho : \Gamma]\!] = ([\![\rho(x_0)]\!], \ldots, [\![\rho(x_{n-1})]\!])
$$

$$\llbracket x \rrbracket \phi \gamma \qquad\qquad \simeq \quad \gamma(x)$$

$$\llbracket r \rrbracket \phi \gamma \qquad\qquad \simeq \quad r \qquad (r \in \mathbb{R})$$

$$\llbracket M + N \rrbracket \phi \gamma \qquad\qquad \simeq \quad \llbracket M \rrbracket \phi \gamma + \llbracket N \rrbracket \phi \gamma$$

$$\llbracket \mathrm{op}(M) \rrbracket \phi \gamma \qquad\qquad \simeq \quad \llbracket \mathrm{op} \rrbracket (\llbracket M \rrbracket \phi \gamma)$$

$$\llbracket \mathtt{let}\ x{:}T = M\ \mathtt{in}\ N \rrbracket \phi \gamma \qquad\qquad \simeq \quad \llbracket N \rrbracket \phi (\gamma [\llbracket M \rrbracket \phi \gamma / x])$$

$$\llbracket * \rrbracket \phi \gamma \qquad\qquad \simeq \quad *$$

$$\llbracket \langle M, N \rangle_{T,U} \rrbracket \phi \gamma \qquad\qquad \simeq \quad \langle \llbracket M \rrbracket \phi \gamma, \llbracket N \rrbracket \phi \gamma \rangle$$

$$\llbracket \mathtt{fst}_{T,U}(M) \rrbracket \phi \gamma \qquad\qquad \simeq \quad \llbracket \pi_0 \rrbracket (\llbracket M \rrbracket \phi \gamma)$$

$$\llbracket \mathtt{snd}_{T,U}(M) \rrbracket \phi \gamma \qquad\qquad \simeq \quad \llbracket \pi_1 \rrbracket (\llbracket M \rrbracket \phi \gamma)$$

$$\llbracket \mathtt{if}\ B\ \mathtt{then}\ M\ \mathtt{else}\ N \rrbracket \phi \gamma \quad \simeq \quad \begin{cases} \llbracket M \rrbracket \phi \gamma & (\llbracket B \rrbracket \phi \gamma \simeq t\!t) \\ \llbracket N \rrbracket \phi \gamma & (\llbracket B \rrbracket \phi \gamma \simeq f\!\!f) \\ \uparrow & (\text{otherwise}) \end{cases}$$

$$\llbracket \mathtt{letrec}\ f(x{:}T){:}U\ =\ M\ \mathtt{in}\ N \rrbracket \phi \gamma \ \simeq \ \begin{array}{l} \llbracket N \rrbracket (\phi[\mu\alpha{:}\mathcal{S}[\llbracket T \rrbracket, \llbracket U \rrbracket]. \\ \quad \lambda a{:}\llbracket T \rrbracket.\ \llbracket M \rrbracket (\phi[\alpha/f])a/f]) \gamma \end{array}$$

$$\llbracket f(M) \rrbracket \phi \gamma \qquad\qquad \simeq \quad \phi(f)(\llbracket M \rrbracket \phi \gamma)$$

$$\llbracket M.\mathrm{rd}_L(x{:}T.\ N) \rrbracket \phi \gamma \qquad\qquad \simeq \quad \mathrm{d}^r_{\llbracket L \rrbracket \phi \gamma}(a \in \llbracket T \rrbracket \mapsto \llbracket N \rrbracket \phi (\gamma[a/x]))(\llbracket M \rrbracket \phi \gamma)$$

$$\llbracket \mathtt{true} \rrbracket \phi \gamma \qquad\qquad \simeq \quad t\!t$$

$$\llbracket \mathtt{false} \rrbracket \phi \gamma \qquad\qquad \simeq \quad f\!\!f$$

$$\llbracket \mathrm{pred}(M) \rrbracket \phi \gamma \qquad\qquad \simeq \quad \llbracket \mathrm{pred} \rrbracket (\llbracket M \rrbracket \phi \gamma)$$

Fig. 7. Denotational semantics

- For every $\varphi{:}\Phi$, where $\Phi = f_0{:}T_0 \to U_n, \ldots, f_{n-1}{:}T_{n-1} \to U_{n-1}$, we define $\llbracket \varphi{:}\Phi \rrbracket \in \llbracket \Phi \rrbracket$ by:
$$\llbracket \varphi{:}\Phi \rrbracket = (\llbracket \varphi(f_0) \rrbracket, \ldots, \llbracket \varphi(f_{n-1}) \rrbracket)$$
- For every $\mathrm{Cl} = \mathbf{clo}_\varphi(f(x{:}T){:}U.\ M){:}T \to U$ we define $\llbracket \mathrm{Cl} \rrbracket \in \mathcal{S}[\llbracket T \rrbracket, \llbracket U \rrbracket]$ by:
$$\llbracket \mathrm{Cl} \rrbracket \quad = \quad \mu\alpha{:}\mathcal{S}[\llbracket T \rrbracket, \llbracket U \rrbracket].\ \lambda a{:}\llbracket T \rrbracket.\llbracket M \rrbracket(\llbracket \varphi{:}\Phi_\varphi \rrbracket[\alpha/f]) a$$

We omit Γ and Φ from $\llbracket \rho{:}\Gamma \rrbracket$ and $\llbracket \varphi{:}\Phi \rrbracket$ when they can be understood from the context.

LEMMA 5.1. *For any type T we have:*

(1) The denotation $\llbracket V \rrbracket$ of any value $V : T$ exists.
(2) For any $v \in \llbracket T \rrbracket$ there is a unique value $V : T$ such that $v = \llbracket V \rrbracket$.

The following two fairly standard results about evaluation contexts are needed to show adequacy.

LEMMA 5.2 (EVALUATION CONTEXT COMPOSITIONALITY). *Suppose* $\Phi \,|\, \Gamma \vdash M : U$, $\Phi \,|\, \Gamma \vdash E[M] : T$, *and* $\Phi \,|\, \Gamma \vdash N : U$. *Then:*

$$\llbracket M \rrbracket \phi \gamma \simeq \llbracket N \rrbracket \phi \gamma \implies \llbracket E[M] \rrbracket \phi \gamma \simeq \llbracket E[N] \rrbracket \phi \gamma \qquad (\phi \in \llbracket \Phi \rrbracket, \gamma \in \llbracket \Gamma \rrbracket)$$

Analogous results hold for: boolean terms in evaluation contexts, $E[B]$; terms in boolean evaluation contexts $E_{\mathsf{bool}}[M]$; and boolean terms in boolean evaluation contexts $E_{\mathsf{bool}}[B]$.

LEMMA 5.3 (EVALUATION CONTEXT STRICTNESS).

(1) *Suppose that* $\Phi \,|\, \Gamma \vdash E[M] : T$ *and* $\Phi \,|\, \Gamma \vdash M : U$. *Then, for any* $x \notin \mathrm{FV}(E)$, *we have:*

$$\llbracket E[M] \rrbracket \phi \gamma \downarrow \implies \llbracket M \rrbracket \phi \gamma \downarrow \quad (\phi \in \llbracket \Phi \rrbracket, \gamma \in \llbracket \Gamma \rrbracket)$$

and so:

$$\llbracket E[M] \rrbracket \phi \gamma \simeq \llbracket E[x] \rrbracket \phi (\gamma [\llbracket M \rrbracket \phi \gamma / x]) \quad (\phi \in \llbracket \Phi \rrbracket, \gamma \in \llbracket \Gamma \rrbracket)$$

The analogous result holds for terms in boolean contexts $E_{\mathsf{bool}}[M]$.

(2) *Suppose that* $\Phi \,|\, \Gamma \vdash E[B] : T$ *and* $\Phi \,|\, \Gamma \vdash B$. *Then:*

$$\llbracket E[B] \rrbracket \phi \gamma \downarrow \implies \llbracket B \rrbracket \phi \gamma \downarrow \quad (\phi \in \llbracket \Phi \rrbracket, \gamma \in \llbracket \Gamma \rrbracket)$$

The analogous result holds for boolean terms in boolean contexts $E_{\mathsf{bool}}[B]$.

6 ADEQUACY

We present our main results, on the correspondence between operational and denotational semantics. Taken together these are our adequacy theorems. As well as the usual correctness and completeness theorems, there are results peculiar to differentiation, both of interest in themselves and also necessary for the others. Theorem 6.1 shows the correctness of our source code transformation of trace terms, in that formal differentiation corresponds to actual differentiation. Theorem 6.2 has two parts. The first is the usual statement of correctness for the ordinary evaluation relation. The second is a statement of correctness of symbolic evaluation. This states that the trace term resulting from symbolic evaluation of a term has the same denotation not only at the environment used for the symbolic evaluation but on a whole open set including it. This, and Theorem 6.1, are needed to prove the ordinary correctness of the ordinary evaluation relation in the case of differentiation as in that case ordinary evaluation proceeds by first symbolically differentiating and then applying our source code transformation. Finally Theorem 6.7 is the expected completeness theorem, that if the semantics of a term is defined, then both its ordinary and symbolic evaluation terminate. Its proof makes use of Proposition 3.5 which interpolates a symbolic evaluation inside any ordinary (non-boolean) evaluation.

6.1 Operational Correctness

THEOREM 6.1 (REVERSE-MODE DIFFERENTIATION). *Suppose that* $\Gamma[x : T] \vdash C : U$, $\Gamma \vdash V : T$ *and* $\Gamma \vdash W : U$ *(and so $\Gamma \vdash W.\mathsf{rd}_V(x : T. C) : T$). Then, for any $\gamma \in \llbracket \Gamma \rrbracket$, we have:*

$$\llbracket W.\mathcal{R}_V(x : T. C) \rrbracket \gamma \simeq \llbracket W.\mathsf{rd}_V(x : T. C) \rrbracket \gamma$$

PROOF. The proof is by structural induction on C. We give a representative case. Suppose C is $\mathtt{let}\ y : U\ =\ D\ \mathtt{in}\ E$. Set $\gamma_a = \gamma[a/x] \in \llbracket \Gamma[x : T] \rrbracket$, for any $a \in \llbracket T \rrbracket$, $\gamma_V = \gamma_{\llbracket V \rrbracket \gamma}$, and $\gamma' = \gamma_V[\llbracket D \rrbracket \gamma_V / y] \in \llbracket \Gamma[x : T][y : U] \rrbracket$. Then

$$\llbracket W.\mathcal{R}_V(x : T. C) \rrbracket \gamma \simeq \llbracket W.\mathcal{R}_V(x : T. E) +_T (\mathtt{let}\ \overline{y} : S\ =\ W.\mathcal{R}_y(y : S. E)\ \mathtt{in}\ \overline{y}.\mathcal{R}_V(x : T. D)) \rrbracket \gamma'$$

with $x \notin \mathrm{FV}(V, W)$, $y \notin \mathrm{FV}(W)$, and $y, \overline{y} \notin \mathrm{FV}(V, D)$. We calculate:

$$\mathrm{d}^r_{[\![V]\!]_\gamma}(a \in [\![T]\!] \mapsto [\![\texttt{let } y{:}U \ = \ D \texttt{ in } E]\!]\gamma_a)[\![W]\!]\gamma$$
$$\simeq \ \mathrm{d}^r_{[\![V]\!]_\gamma}(a \in [\![T]\!] \mapsto [\![E]\!]\gamma_a[[\![D]\!]\gamma_a/y])[\![W]\!]\gamma$$
$$\simeq \ \mathrm{d}^r_{[\![V]\!]_\gamma}((c \in [\![T \times U]\!] \mapsto [\![E]\!]\gamma_{\pi_0(c)}[\pi_1(c)/y]) \circ \langle a \in [\![T]\!] \mapsto a, \ a \in [\![T]\!] \mapsto [\![D]\!]\gamma_a\rangle)[\![W]\!]\gamma$$
$$\simeq \ \mathrm{d}^r_{[\![V]\!]_\gamma}(\langle a \in [\![T]\!] \mapsto a, \ a \in [\![T]\!] \mapsto [\![D]\!]\gamma_a\rangle[$$
$$(\mathrm{d}^r_{\langle[\![V]\!]_\gamma,[\![D]\!]\gamma_V\rangle}(c \in [\![T \times U]\!] \mapsto [\![E]\!]\gamma_{\pi_0(c)}[\pi_1(c)/y]))[\![W]\!]\gamma]$$
$$\simeq \ \mathrm{d}^r_{[\![V]\!]_\gamma}(\langle a \in [\![T]\!] \mapsto a, \ a \in [\![T]\!] \mapsto [\![D]\!]\gamma_a\rangle[$$
$$\langle\mathrm{d}^r_{[\![V]\!]_\gamma}(a \in [\![T]\!] \mapsto [\![E]\!]\gamma_a[[\![D]\!]\gamma_V/y]), \mathrm{d}^r_{[\![D]\!]\gamma_V}(b \in [\![U]\!] \mapsto [\![E]\!]\gamma_{[\![V]\!]_\gamma}[b/y])\rangle[\![W]\!]\gamma]$$
$$\simeq \ \mathrm{d}^r_{[\![V]\!]_\gamma}(\langle a \in [\![T]\!] \mapsto a, \ a \in [\![T]\!] \mapsto [\![D]\!]\gamma_a\rangle[$$
$$\langle\mathrm{d}^r_{[\![V]\!]_\gamma}(a \in [\![T]\!] \mapsto [\![E]\!]\gamma_a[[\![D]\!]\gamma_V/y])[\![W]\!]\gamma, \mathrm{d}^r_{[\![D]\!]\gamma_V}(b \in [\![U]\!] \mapsto [\![E]\!]\gamma_{[\![V]\!]_\gamma}[b/y])[\![W]\!]\gamma\rangle]$$
$$\simeq \ \mathrm{d}^r_{[\![V]\!]_{\gamma'}}(a \in [\![T]\!] \mapsto a)[\mathrm{d}^r_{[\![V]\!]_\gamma}(a \in [\![T]\!] \mapsto [\![E]\!]\gamma_a[[\![D]\!]\gamma_V/y])[\![W]\!]\gamma]$$
$$+ \ \mathrm{d}^r_{[\![V]\!]_{\gamma'}}(a \in [\![T]\!] \mapsto [\![D]\!]\gamma')[\mathrm{d}^r_{[\![D]\!]\gamma_V}(b \in [\![U]\!] \mapsto [\![E]\!]\gamma_{[\![V]\!]_\gamma}[b/y])[\![W]\!]\gamma]$$
$$\simeq \ \mathrm{d}^r_{[\![V]\!]_\gamma}(a \in [\![T]\!] \mapsto [\![E]\!]\gamma_a[[\![D]\!]\gamma_V/y])[\![W]\!]\gamma$$
$$+ \ \mathrm{d}^r_{[\![V]\!]_{\gamma'}}(a \in [\![T]\!] \mapsto [\![D]\!]\gamma')[\mathrm{d}^r_{[\![D]\!]\gamma_V}(b \in [\![U]\!] \mapsto [\![E]\!]\gamma_{[\![V]\!]_\gamma}[b/y])[\![W]\!]\gamma]$$
$$\simeq \ [\![W.\mathcal{R}_V(x{:}T.E) +_T (\texttt{let } \overline{y}{:}S \ = \ W.\mathcal{R}_y(y{:}S.E) \texttt{ in } \overline{y}.\mathcal{R}_V(x{:}T.D))]\!]\gamma'$$

Note the use of Equation (10) in the the fourth step. □

THEOREM 6.2 (OPERATIONAL CORRECTNESS). *Suppose that* $\Phi \mid \Gamma \vdash M{:}T, \vdash \varphi{:}\Phi$, *and* $\vdash \rho : \Gamma$. *Then:*

(1) **Operational semantics.**

$$\varphi \mid \rho \vdash M \Rightarrow V \implies [\![M]\!][\![\varphi]\!][\![\rho]\!] = [\![V]\!]$$

(and similarly for boolean terms).

(2) **Symbolic operational semantics.**

$$\varphi \mid \rho \vdash M \rightsquigarrow C \implies \exists O \subseteq_{open} [\![\Gamma]\!]. [\![\rho]\!] \in O \wedge \forall \gamma \in O. [\![M]\!][\![\varphi]\!]\gamma \simeq [\![C]\!]\gamma$$

PROOF. The two parts are proved by mutual induction on the size of the proofs that establish the given operational relations, and by cases on the form of M. As an example case of the second part, suppose M has the form $\texttt{let } x : T \ = \ V \texttt{ in } L$. Then for some D and V' we have a smaller proof of $\varphi \mid \rho[V'/x] \vdash L \rightsquigarrow D$, where $V' = \rho(V)$, and C has the form $\texttt{let } x : T \ = \ V \texttt{ in } D$.

By the induction hypothesis there is an open set O such that $[\![\rho[V'/x]]\!] \in O$ and, for all $\delta \in O$, we have $[\![L]\!][\![\varphi]\!]\delta \simeq [\![D]\!]\delta$. Set $\theta = \gamma \in [\![\Gamma]\!] \mapsto \gamma[[\![V]\!]\gamma/x]$. As θ is continuous, $O' =_{\mathrm{def}} \theta^{-1}(O)$ is open. We show it is the required open set.

- First, $[\![\rho]\!] \in O'$ as we have: $\theta([\![\rho]\!]) = [\![\rho]\!][[\![V]\!][\![\rho]\!]/x] = [\![\rho]\!][[\![V']\!]/x] = [\![\rho[V'/x]]\!] \in O$.
- Second, for any $\gamma \in O'$ we have:

$$\begin{aligned}
[\![\texttt{let } x : T \ = \ V \texttt{ in } L]\!][\![\varphi]\!]\gamma &\simeq \ [\![L]\!][\![\varphi]\!]\gamma[[\![V]\!]\gamma/x] \\
&\simeq \ [\![D]\!]\gamma[[\![V]\!]\gamma/x] \qquad\qquad (\text{as } \theta(\gamma) \in O) \\
&\simeq \ [\![\texttt{let } x : T \ = \ V \texttt{ in } D]\!]\gamma
\end{aligned}$$

□

The following corollary shows that our strategy of first symbolically reducing to produce a trace term, then symbolically differentiating, is correct.

COROLLARY 6.3. *Suppose that* $\Phi \mid \Gamma[x{:}T] \vdash M{:}U$, $\Gamma \vdash V{:}T$, *and* $\Gamma \vdash W{:}U$. *Then, for any* $\vdash \varphi : \Phi$ *and* $\vdash \rho : \Gamma$ *we have:*

$$\varphi \mid \rho[V\rho/x] \vdash M \rightsquigarrow C \implies [\![W.\mathrm{rd}_V(x : T. M)]\!][\![\varphi]\!][\![\rho]\!] \simeq [\![W.\mathrm{rd}_V(x : T. C)]\!][\![\varphi]\!][\![\rho]\!]$$

6.2 Operational Completeness

We turn to proving operational completeness, that the evaluation, or symbolic evaluation, of a term terminates when it should, i.e., when its denotation is defined. For terms M write $\varphi \mid \rho \vdash M \Downarrow$ when, for some V, $\varphi \mid \rho \vdash M \Rightarrow V$ and, assuming φ and ρ known from the context, say that M *terminates*; we adopt similar terminology for boolean terms.

To prove operational completeness we use a standard strategy: first proving operational completeness for an auxiliary "approximation language" in which recursive definitions are replaced by approximations to them, which we call *limited recursive definitions*, and then lifting that result to the main language. Specifically we replace the syntactic form for recursive definitions of the main language by the family of syntactic forms:

$$\texttt{letrec}_n f(x:T):U \;=\; M \texttt{ in } N \qquad (n \in \mathbb{N})$$

with the evident typing rule, and make analogous consequential changes in the various definitions.

In the definition of function environments and closures, the clause for closures becomes:

- If $\mathrm{FFV}(M)\backslash\{f\} \subseteq \mathrm{Dom}(\varphi)$, $\mathrm{FV}(M) \subseteq \{x\}$, and $n \in \mathbb{N}$, then

$$\langle n, \varphi, f, x, T, U, M \rangle$$

is a closure, written as: $\mathbf{clo}_{n,\varphi}(f(x:T):U.\,M)$.

The limited recursive definition redexes are $\texttt{letrec}_n f(x{:}T){:}U \;=\; M \texttt{ in } N$; their evaluation rules are in Figure 8. There, and below, we write $\vdash_l$ for limited recursion language judgements.

$$\frac{\varphi[\mathbf{clo}_{n,\varphi}(f(x{:}T){:}U.\,M)/f]|\rho \vdash_l N \Rightarrow V'}{\varphi|\rho \vdash_l \texttt{letrec}_n f(x{:}T){:}U = M \texttt{ in } N \Rightarrow V'}$$

$$\frac{\varphi|\rho \vdash_l V \Rightarrow V' \quad \varphi'[\mathbf{clo}_{n,\varphi'}(f(x{:}T){:}U.\,M)/f]|\{x \mapsto V'\} \vdash_l M \Rightarrow W}{\varphi|\rho \vdash_l f(V) \Rightarrow W} \qquad \text{where } \varphi(f) = \mathbf{clo}_{n+1,\varphi'}(f(x:T):U.\,M)$$

$$\frac{\varphi[\mathbf{clo}_{n,\varphi}(f(x{:}T){:}U.\,M)/f]|\rho \vdash_l N \rightsquigarrow C}{\varphi|\rho \vdash_l \texttt{letrec}_n f(x{:}T){:}U = M \texttt{ in } N \rightsquigarrow C}$$

$$\frac{\varphi|\rho \vdash_l V \Rightarrow V' \quad \varphi'[\mathbf{clo}_{n,\varphi'}(f(x{:}T){:}U.\,M)/f]|\{x \mapsto V'\} \vdash_l M \rightsquigarrow C}{\varphi|\rho \vdash_l f(V) \rightsquigarrow \texttt{let } x{:}T = V \texttt{ in } C} \qquad \text{where } \varphi(f) = \mathbf{clo}_{n+1,\varphi'}(f(x:T):U.\,M)$$

Fig. 8. Operational semantics of bounded recursion

For the closure typing judgement $\mathrm{Cl}{:}T \rightarrow U$ we substitute:

$$\frac{\vdash_l \varphi : \Phi \quad \Phi[f{:}T \rightarrow U] \mid x{:}T \vdash_l M{:}U}{\vdash_l \mathbf{clo}_{n,\varphi}(f(x:T):U.\,M) : T \rightarrow U}$$

As regards the denotational semantics, the clause for limited recursive definitions is:

$$[\![\texttt{letrec}_n f(x{:}T){:}U \;=\; M \texttt{ in } N]\!](\phi)(\gamma) \;\simeq\;$$
$$[\![N]\!](\phi[(\mu_n\alpha{:}\mathcal{S}[\![T]\!],[\![U]\!].\,\lambda a{:}[\![T]\!].\,[\![M]\!](\phi[\alpha/f])\,a)/f])(\gamma)$$

The results for the operational and denotational semantics carry over to the restricted setting, and we refer to them in the same way as we do to the unrestricted versions.

Relating the two languages, the *n-th approximant* $M^{(n)}$ of a language term M is obtained by replacing every recursive definition in M by an n-limited one (similarly for boolean terms) and n-th approximants of closures and function environments are defined by structural recursion:

$$\mathbf{clo}_\varphi(f(x{:}T){:}U.\,M)^{(n)} \;\;=\;\; \mathbf{clo}_{n,\varphi^{(n)}}(f(x{:}T){:}U.\,M^{(n)})$$

$$\{\ldots, f_i \mapsto \mathrm{Cl}_i, \ldots\}^{(n)} = \{\ldots, f_i \mapsto \mathrm{Cl}_i^{(n)}, \ldots\}$$

The terms $M^{(n)}$ can be defined by structural recursion; we just give one clause of the definition:

$$(\texttt{letrec } f(x:T):U = M \texttt{ in } N)^{(n)} = \texttt{letrec}_n f(x:T):U = M^{(n)} \texttt{ in } N^{(n)}$$

Approximation preserves typing judgments.

Termination is proved for the approximation language by structural induction via a suitable notion of *computability*.

- A closure

$$\vdash_l \mathbf{clo}_{n,\varphi}(f(x:T):U.\,M):T \to U$$

is computable iff $n = 0$ or $n > 0$ and, for all $\vdash_l V : T$ we have:

$$[\![M]\!][\![(\varphi[\mathbf{clo}_{n-1,\varphi}(f(x:T):U.\,M)/f]):\Phi_\varphi[f:T \to U]]\!][\![V]\!]\!\downarrow \implies$$
$$\varphi[\mathbf{clo}_{n-1,\varphi}(f(x:T):U.\,M)/f]\,|\,\{x \mapsto V\} \vdash_l M\!\Downarrow$$

- A function environment $\vdash_l \varphi : \Phi$ is computable iff $\vdash_l \varphi(f) : \Phi(f)$ is a computable closure, for every $f \in \mathrm{Dom}(\varphi)$.
- A term $\Phi \,|\, \Gamma \vdash_l M : T$ is computable iff for every computable $\vdash_l \varphi : \Phi$ and every $\vdash \rho : \Gamma$

$$[\![M]\!][\![\varphi]\!][\![\rho]\!]\downarrow \implies \varphi \,|\, \rho \vdash_l M\!\Downarrow$$

(and similarly for boolean terms).

Strictly speaking, in the above we should say that it is the sequent $\Phi \,|\, \Gamma \vdash_l M : T$ that is computable and similarly for closures and function environments.

LEMMA 6.4.
(1) Every closure $\vdash_l \mathbf{clo}_{n,\varphi}(f(x:T):U.\,M):T \to U$ is computable.
(2) Every function environment $\vdash_l \varphi : \Phi$ is computable.
(3) Every term $\Phi \,|\, \Gamma \vdash_l M : T$ is computable.
(4) Every boolean term $\Phi \,|\, \Gamma \vdash_l B$ is computable.

The next two lemmas enable us to lift completeness from the approximation language to the main one. The first lets us pass from semantic existence in the main language to semantic existence in the approximation language; the second allows us to pass in the opposite direction from termination in the approximation language to termination in the main one.

LEMMA 6.5. *For any well-typed term M of the main language we have:*

$$[\![M]\!] = \bigvee_{n \in \mathbb{N}} [\![M^{(n)}]\!]$$

and similarly for boolean terms, closures, and function environments.

LEMMA 6.6. *For any term M of the main language we have:*

$$\varphi^{(n)} \,|\, \rho \vdash_l M^{(n)} \Rightarrow V \implies \varphi \,|\, \rho \vdash M \Rightarrow V$$

and

$$\varphi^{(n)} \,|\, \rho \vdash_l M^{(n)} \rightsquigarrow C \implies \varphi \,|\, \rho \vdash M \rightsquigarrow C$$

and similarly for boolean terms.

Operational completeness follows straightforwardly from these three lemmas:

Theorem 6.7 (Operational completeness). *Suppose that* $\Phi \mid \Gamma \vdash M : T, \vdash \varphi : \Phi$, *and* $\vdash \rho : \Gamma$. *Then:*

(1) ***Operational semantics.***

$$[\![M]\!][\![\varphi]\!][\![\rho]\!] \downarrow \implies \varphi \mid \rho \vdash M \Downarrow$$

   (and similarly for boolean terms).

(2) ***Symbolic operational semantics.***

$$[\![M]\!][\![\varphi]\!][\![\rho]\!] \downarrow \implies \exists C. \varphi \mid \rho \vdash M \rightsquigarrow C$$

Proof.

(1) Suppose $[\![M]\!][\![\varphi]\!][\![\rho]\!] \downarrow$. By Lemma 6.5 $[\![M]\!][\![\varphi]\!] = \bigvee_n [\![M^{(n)}]\!][\![\varphi^{(n)}]\!]$. So $[\![M^{(n)}]\!][\![\varphi^{(n)}]\!][\![\rho]\!] \downarrow$ for some n. By Lemma 6.4, both $M^{(n)}$ and $\varphi^{(n)}$ are computable. So $\varphi^{(n)} \mid \rho \vdash_I M^{(n)} \Downarrow$. By Lemma 6.6, we then have $\varphi \mid \rho \vdash M \Downarrow$, as required. The proof for boolean terms is similar.

(2) This follows from the first part and Proposition 3.5.

$\square$

7 DISCUSSION

There is much more to do on the theory of differentiable programming languages, even at a basic level; we briefly suggest some possibilities. Trace-based automatic differentiation systems generally work with A-normal forms, or equivalent structures. They also employ optimizations. For example, as mentioned in the Introduction, they may record auxiliary information in evaluation traces to reduce recomputation. Other automatic differentiation systems rely on code transformations for differentiation. It would be interesting to define and study such optimizations and alternative approaches, perhaps in the setting of our language.

Another interesting possibility would be to work with non-differentiable functions like ReLU or with non-smooth functions. For the former, one might use Clarke sub-gradients [Clarke 1990], following [Di Gianantonio and Edalat 2013] (the Clarke sub-gradient of ReLU at 0 is the interval $[0, 1]$); for the latter, one may use C^k functions. Yet another possibility would be to work with approximate reals, rather than reals, and to seek numerical accuracy theorems; one might employ a domain-theoretic notion of sub-differentiation of functions over Scott's interval domain, generalizing the Clarke sub-gradient (see [Edalat and Lieutier 2004; Edalat and Maleki 2018]).

One would like results for richer languages, with a wider range of types or with computational effects. The problem is then how these additional features interact with differentiation. An extension to side-effects would make contact with the literature on the automatic differentiation of imperative languages, such as Fortran. An extension to probability, in some form, would make contact with stochastic optimization and, further, with probabilistic languages for statistical learning. For higher-order types, there may be a domain-theoretic analogue of convenient vector spaces that additionally supports recursion. Further, one might, as suggested in [Vákár et al. 2018], seek a domain-theoretic analogue of diffeological spaces (see [Iglesias-Zemmour 2013]); that would also accommodate sum and recursive types. One might also wish to program with Riemannian manifolds, to accommodate natural gradient descent [Amari 1996]; these too should fit into a diffeological framework. In another direction, the work on categories with differential structure may yield an axiomatic version of adequacy theorems for programming languages with differentiation constructs; such categories further equipped with structure to model partiality [Cockett et al. 2011] are of particular interest.

Finally, if perhaps orthogonally, it is important to add explicit tensor (multi-dimensional array) types, with accompanying shape analysis. There is a long history of programming-language design in this area; a salient example is the design of Remora [Slepak et al. 2014].

REFERENCES

Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016a. TensorFlow: a system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, 265–283.

Martin Abadi, Andy Chu, Ian Goodfellow, H. Brendan McMahan, Ilya Mironov, Kunal Talwar, and Li Zhang. 2016b. Deep learning with differential privacy. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*. ACM, 308–318. https://doi.org/10.1145/2976749.2978318

Samson Abramsky and Achim Jung. 1994. Domain theory. In *Handbook of Logic in Computer Science (Vol. 3)*, Samson Abramsky, Dov M. Gabbay, and T. S. E. Maibaum (Eds.). Oxford University Press, Inc., 1–168.

Akshay Agrawal, Akshay Naresh Modi, Alexandre Passos, Allen Lavoie, Ashish Agarwal, Asim Shankar, Igor Ganichev, Josh Levenberg, Mingsheng Hong, Rajat Monga, et al. 2019. TensorFlow Eager: A multi-stage, Python-embedded DSL for machine learning. *arXiv preprint arXiv:1903.01855* (2019).

Shun-ichi Amari. 1996. Neural learning in structured parameter spaces — natural Riemannian gradient. In *Advances in Neural Information Processing Systems 9, NIPS*, M. Mozer, M. I. Jordan, and T. Petsche (Eds.). MIT Press, 127–133.

Atilim Günes Baydin, Barak A. Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. 2018. Automatic differentiation in machine learning: a survey. *Journal of Machine Learning Research* 18, 153 (2018), 1–43.

Atilim Günes Baydin, Barak A. Pearlmutter, and Jeffrey Mark Siskind. 2016. Tricks from deep learning. *CoRR* abs/1611.03777 (2016).

Thomas Beck and Herbert Fischer. 1994. The if-problem in automatic differentiation. *J. Comput. Appl. Math.* 50, 1-3 (May 1994), 119–131. https://doi.org/10.1016/0377-0427(94)90294-1

James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley, and Yoshua Bengio. 2010. Theano: A CPU and GPU math expression compiler. In *Proceedings of the Python for scientific computing conference (SciPy)*, Vol. 4.

Yves Bertot and Pierre Castéran. 2013. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer Science & Business Media.

Richard Blute, Thomas Ehrhard, and Christine Tasson. 2010. A convenient differential category. *arXiv preprint arXiv:1006.3140* (2010).

Richard F Blute, J Robin B Cockett, and Robert AG Seely. 2009. Cartesian differential categories. *Theory and Applications of Categories* 22, 23 (2009), 622–672.

Antonio Bucciarelli, Thomas Ehrhard, and Giulio Manzonetto. 2010. Categorical models for simply typed resource calculi. *Electronic Notes in Theoretical Computer Science* 265 (2010), 213–230.

Bruce Christianson. 2012. A Leibniz notation for automatic differentiation. In *Recent Advances in Algorithmic Differentiation*, Shaun Forth, Paul Hovland, Eric Phipps, Jean Utke, and Andrea Walther (Eds.). Lecture Notes in Computational Science and Engineering, Vol. 87. Springer, 1–9.

Frank H. Clarke. 1990. *Optimization and nonsmooth analysis*. Classics in Applied Mathematics, Vol. 5. SIAM.

J Robin B Cockett, Geoff SH Cruttwell, and Jonathan D Gallagher. 2011. Differential restriction categories. *Theory and Applications of Categories* 25, 21 (2011), 537–613.

Leonardo Mendonça de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. 2015. The Lean Theorem Prover (System Description). In *Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings (Lecture Notes in Computer Science)*, Amy P. Felty and Aart Middeldorp (Eds.), Vol. 9195. Springer, 378–388. https://doi.org/10.1007/978-3-319-21401-6_26

Pietro Di Gianantonio and Abbas Edalat. 2013. A language for differentiable functions. In *Foundations of Software Science and Computation Structures*, Frank Pfenning (Ed.). Springer, 337–352.

Abbas Edalat and André Lieutier. 2004. Domain theory and differential calculus (functions of one variable). *Mathematical Structures in Computer Science* 14, 6 (2004), 771–802. https://doi.org/10.1017/S0960129504004359

Abbas Edalat and Mehrdad Maleki. 2018. Differential calculus with imprecise input and its logical framework. In *Foundations of Software Science and Computation Structures - 21st International Conference, FOSSACS 2018*. 459–475. https://doi.org/10.1007/978-3-319-89366-2_25

Thomas Ehrhard and Laurent Regnier. 2003. The differential lambda-calculus. *Theo. Comp. Sci.* 309, 1-3 (2003), 1–41.

Conal Elliott. 2018. The simple essence of automatic differentiation. In *Proceedings of the ACM on Programming Languages (ICFP)*.

Matthias Felleisen and Daniel P. Friedman. 1987. Control operators, the SECD-machine and the λ-calculus. In *Formal Description of Programming Concepts III*, M. Wirsing (Ed.). Elsevier, 193–217.

H. Fischer. 2001. Automatic differentiation: root problem and branch problem. In *Encyclopedia of Optimization*, C. A. Floudas and P. M. Pardalos (Eds.). Vol. I. Kluwer Academic Publishers, 118–122.

Roy Frostig, Matthew James Johnson, and Chris Leary. 2018. Compiling machine learning programs via high-level tracing. Presented at SysML 2018.

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep Learning*. MIT Press.

Andreas Griewank. 2000. *Evaluating derivatives - principles and techniques of algorithmic differentiation*. Frontiers in applied mathematics, Vol. 19. SIAM.

L. Hascoët and V. Pascual. 2013. The Tapenade automatic differentiation tool: principles, model, and specification. *ACM Transactions On Mathematical Software* 39, 3 (2013).

P. Iglesias-Zemmour. 2013. *Diffeology*. American Mathematical Society.

Andreas Kriegl and Peter W Michor. 1997. *The convenient setting of global analysis*. Vol. 53. American Mathematical Soc.

Dougal Maclaurin, David Duvenaud, and Ryan P Adams. 2015. Autograd: effortless gradients in Numpy. In *ICML 2015 AutoML Workshop*, Vol. 238.

Oleksandr Manzyuk. 2012. A simply typed λ-calculus of forward automatic differentiation. *Electronic Notes in Theoretical Computer Science* 286 (2012), 257 − 272. https://doi.org/10.1016/j.entcs.2012.08.017 Proceedings of the 28th Conference on the Mathematical Foundations of Programming Semantics (MFPS XXVIII).

Micaela Mayero. 2002. Using theorem proving for numerical analysis (correctness proof of an automatic differentiation algorithm). In *Theorem Proving in Higher Order Logics, 15th International Conference, TPHOLs 2002, Hampton, VA, USA, August 20-23, 2002, Proceedings (Lecture Notes in Computer Science)*, Victor Carreño, César A. Muñoz, and Sofiène Tahar (Eds.), Vol. 2410. Springer, 246−262. https://doi.org/10.1007/3-540-45685-6_17

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. PyTorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*. 8024−8035.

Barak A. Pearlmutter and Jeffrey Mark Siskind. 2008. Reverse-mode AD in a functional framework: lambda the ultimate backpropagator. *ACM Trans. Program. Lang. Syst.* 30, 2 (2008), 7:1−7:36. https://doi.org/10.1145/1330017.1330018

Daniel Selsam, Percy Liang, and David L. Dill. 2017. Developing bug-free machine learning systems with formal mathematics. In *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017 (Proceedings of Machine Learning Research)*, Doina Precup and Yee Whye Teh (Eds.), Vol. 70. PMLR, 3047−3056.

Amir Shaikhha, Andrew Fitzgibbon, Dimitrios Vytiniotis, Simon Peyton Jones, and Christoph Koch. 2018. Efficient differentiable programming in a functional array-processing language. *CoRR* abs/1806.02136 (2018). arXiv:1806.02136

Jeffrey Mark Siskind and Barak A. Pearlmutter. 2005. Perturbation confusion and referential transparency: correct functional implementation of forward-mode AD. In *Implementation and Application of Functional Languages—17th International Workshop, IFL'05*, A. Butterfield (Ed.). 1−9. Trinity College Dublin, Computer Science Department Technical Report TCD-CS-2005-60.

Jeffrey Mark Siskind and Barak A. Pearlmutter. 2008. Nesting forward-mode AD in a functional framework. *Higher-Order and Symbolic Computation* 21, 4 (2008), 361−376. https://doi.org/10.1007/s10990-008-9037-1

Justin Slepak, Olin Shivers, and Panagiotis Manolios. 2014. An array-oriented language with static rank polymorphism. In *Proceedings of the 23rd European Symposium on Programming Languages and Systems - Volume 8410*. Springer-Verlag, 27−46. https://doi.org/10.1007/978-3-642-54833-8_3

Shuang Song, Kamalika Chaudhuri, and Anand D. Sarwate. 2013. Stochastic gradient descent with differentially private updates. In *IEEE Global Conference on Signal and Information Processing, GlobalSIP 2013, Austin, TX, USA, December 3-5, 2013*. IEEE, 245−248. https://doi.org/10.1109/GlobalSIP.2013.6736861

Seiya Tokui, Kenta Oono, Shohei Hido, and Justin Clayton. 2015. Chainer: a next-generation open source framework for deep learning. In *Proceedings of Workshop on Machine Learning Systems (LearningSys) in the Twenty-Ninth Conference on Neural Information Processing Systems (NIPS)*, Vol. 5. 1−6.

W.F. Trench. 2003. *Introduction to Real Analysis*. Prentice Hall/Pearson Education.

M. Vákár, O. Kammar, and S. Staton. 2018. Diffeological spaces and semantics for differential programming. Presented at Domains XIII Workshop.

Bart van Merrienboer, Dan Moldovan, and Alexander B. Wiltschko. 2018. Tangent: automatic differentiation using source-code transformation for dynamically typed array programming. In *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, 3-8 December 2018, Montréal, Canada.*, S. Bengio, H. M. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett (Eds.). 6259−6268.

Fei Wang, Xilun Wu, Grégory M. Essertel, James M. Decker, and Tiark Rompf. 2018. Demystifying differentiable programming: shift/reset the penultimate backpropagator. *CoRR* abs/1803.10228 (2018). arXiv:1803.10228

Yuan Yu, Martín Abadi, Paul Barham, Eugene Brevdo, Mike Burrows, Andy Davis, Jeff Dean, Sanjay Ghemawat, Tim Harley, Peter Hawkins, Michael Isard, Manjunath Kudlur, Rajat Monga, Derek Murray, and Xiaoqiang Zheng. 2018. Dynamic control flow in large-scale machine learning. In *Proceedings of the Thirteenth EuroSys Conference (EuroSys '18)*. ACM, Article 18, 15 pages. https://doi.org/10.1145/3190508.3190551