Parallelism-Preserving Automatic Differentiation for Second-Order Array Languages

Adam Paszke
apaszke@google.com
Google Research
Poland

Matthew J. Johnson
mattjj@google.com
Google Research
USA

Roy Frostig
frostig@google.com
Google Research
USA

Dougal Maclaurin
dougalm@google.com
Google Research
USA

Abstract

We develop automatic differentiation (AD) procedures for reductions and scans—parameterized by arbitrary differentiable monoids—in a way that preserves parallelism, by rewriting them as other reductions and scans. This is in contrast with the literature and with existing AD systems, which are either general, but force sequential execution of the derivative program, or only include hand-crafted rules for a select few monoids (usually $(0, +)$, $(1, \times)$, $(-\infty, \max)$ and $(\infty, \min)$) and thus lack the general flexibility of second-order languages.

CCS Concepts: • **Software and its engineering → Parallel programming languages**; • **Mathematics of computing → Automatic differentiation**.

Keywords: Automatic differentiation, second-order array languages, scan, reduce

ACM Reference Format:
Adam Paszke, Matthew J. Johnson, Roy Frostig, and Dougal Maclaurin. 2021. Parallelism-Preserving Automatic Differentiation for Second-Order Array Languages. In *Proceedings of the 9th ACM SIGPLAN International Workshop on Functional High-Performance and Numerical Computing (FHPNC '21), August 22, 2021, Virtual, Republic of Korea.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3471873.3472975

1 Introduction

Although the popularization of automatic differentiation (AD) has been transformative for the world of array computing, especially in machine learning, its typical application in practice remains limited to first-order array (domain specific) languages, such as the expressions over the operations found in NumPy or MATLAB. By *first-order*, we mean that the algebra of array operations is fixed, and there exist no array operations parameterized by user-defined functions.

Second-order languages, by contrast, do include primitive array operators parameterized by user functions. A popular such operation is **map**, which applies a user-defined function to every element of the array independently. Another is **reduce**, which, given an array and a monoid structure over the type of array elements, applies the monoidal binary operation repeatedly to combine all elements into a single output. (A related operation is **scan**, which we cover in later sections.) A third is **zip**, which takes two arrays and collates their respective elements into a single array of pairs. Finally, **transpose** exchanges the order of the two leading dimensions of an array. Given these ingredients, matrix multiplication is no longer needed as a language primitive, and can be expressed as:

$$\begin{aligned}&\lambda x : [n][k]\mathsf{Float}.\ \lambda y : [k][m]\mathsf{Float}.\\ &\quad \mathbf{map}(x, \lambda x_{\text{row}} : [k]\mathsf{Float}.\\ &\qquad \mathbf{map}(\mathbf{transpose}(y), \lambda y_{\text{col}} : [k]\mathsf{Float}.\\ &\qquad\quad \mathbf{reduce}_{0,+}(\mathbf{map}(\mathbf{zip}(x_{\text{row}}, y_{\text{col}}), \times))))\end{aligned}$$

Writing matrix multiplication in these terms (particularly **map** and **reduce**) additionally clearly highlights *parallel structure* in its computation, especially following **map**'s data parallelism and **reduce**'s invariance to association order of the monoid operation (+) [5, 6].

There is a wide variety of matrix-multiplication-like operations, such as vector-vector products, matrix-vector products, batched matrix products, and so on. These operations can moreover be defined over a semiring other than $(0, 1, +, \times)$, such as the tropical semiring $(0, \infty, +, \min)$. First-order languages must offer each operation, over each semiring etc., as a distinct primitive. Meanwhile a second-order language with **map** and **reduce** subsumes them all.

So far, to our knowledge, no second-order languages implemented in practice offer first-class AD support (in both

FHPNC '21, August 22, 2021, Virtual, Republic of Korea

ACM ISBN 978-1-4503-8614-2/21/08.
https://doi.org/10.1145/3471873.3472975

forward- and reverse-modes), inhibiting their use in domains that lean heavily on this feature—most notably programs using gradient-based optimization methods. Implementations have essentially no theory to draw from here: there is no guidance or proposed approach in the AD literature for differentiating arbitrary associative reductions and prefix sums. All treatment suggests either handling only first-order array programs, or enforcing a sequential ordering in the derivative program. Either outcome sacrifices one of the strongest features of array programming languages.

In this work, we aim to address this shortcoming by outlining a generalization of AD, as applied in the first-order realm, to a language of parallel second-order operations. Importantly, we propose differentiation rules that preserve some natural degree of parallelism by means of self-reduction, in the sense that AD of **reduce** and **scan** results in other reductions and scans. Turning to limitations, we highlight a sense in which the expressions output by our AD rules incur some efficiency loss (in worst-case runtime and memory) relative to standard expectations generalized from the first-order setting. Closing this gap sets up technical questions for future study: can we write a more efficient derivative for **scan**, or is there a related primitive that is even better suited for AD?

Background and Related Work. Research into how increasingly expressive language features interact with AD is active and ongoing. Formal accounts of AD have been proposed, and proven sound, in settings ranging from minimal first-order languages up to higher-order and Turing complete ones [3, 8, 9, 17, 22, 23].

Our focus is on a more specific desideratum in second-order under AD: beyond establishing sound capability, our concern is *parallelism-preserving* AD and its impacts on efficiency. Our setup involves a particular second-order language that explicitly includes **reduce** and **scan** as primitives, and our goal is to design AD rules that maximally preserve the parallel and associative structure of these operations.

Parallel-aware AD has remained largely unexplored in a formal context to our knowledge, but has naturally been confronted in the course of several system implementations [19, 24]. Regarding reduction and scan in particular, JAX [7, 10] explicitly provides both as primitive operations, and Dex [15, 19] allows for a monoid accumulation effect to be used with its array-constructing **for** expression. Many prior systems, such as TensorFlow [2] and PyTorch [18], or MPI (with AD support explored in Hovland and Bischof [14]) offer special cases of **reduce** under a few basic monoids (e.g. sum and max) but forgo the generic second-order function that accepts an arbitrary differentiable monoid.

Independently of AD, parallelism via **map**-, **reduce**-, and **scan**-like operations—and how to use these in writing efficient numerical programs—is a well-studied topic [5, 6].

Types

t, a, b	::= Float	floating-point number type
	$\mid [n]t$	array type
	$\mid (a, b)$	pair type
	$\mid a \to b$	function type

Expressions

u, w	::= x	variable
	$\mid l$	literal
	$\mid [u, w]$	pair constructor
	$\mid$ fst	left pair projection
	$\mid$ snd	right pair projection
	$\mid \lambda x\!:\!t.\ u$	lambda abstraction
	$\mid u(w)$	application
	$\mid$ let $x\!:\!t = u$ in w	let binding
	$\mid$ **jvp** u	forward-mode AD
	$\mid$ **vjp** u	reverse-mode AD
	$\mid$ Zero t	symbolic zero for vector space t
	$\mid$ **map**	$[n]a \to (a \to b) \to [n]b$
	$\mid$ **reduce**	Monoid $a \to [n]a \to a$
	$\mid$ **scan**	LeftMonoid $a \to [n]a \to [n]a$
	$\mid$ **broadcast**	$a \to [n]a$
	$\mid$ **zip**	$[n]a \to [n]b \to [n](a, b)$
	$\mid$ **reverse**	$[n]a \to [n]a$
	$\mid$ **roll_right**	$a \to [n]a \to ([n]a, a)$
	$\mid$ **transpose**	$[n][m]a \to [m][n]a$

Figure 1. A small second-order array language we will use in this work. Monoid a and LeftMonoid a is syntactic sugar for $(a, (a, a) \to a)$ that additionally implies a (compiler unchecked) invariant that the inputs do obey (a version of) monoidal laws. In particular, in a Monoid a, the first component is the neutral element of the associative binary operation in the second component. LeftMonoid a relaxes the neutral element restriction by allowing the first component to only be a left identity (but not necessarily a right identity).

2 Language

The array language we work with is defined in Figure 1. We omit its typing rules, as the type system we use is conventional. For convenience, all array combinators (presented at the bottom of valid expressions in Figure 1) are annotated with their type signatures. In the remainder we will often skip the explicit type signatures whenever they are clear from the context. Note that the type of **broadcast** as presented here requires use of a type-inference scheme to decide on the size of the returned array.

Again, as their semantics are conventional, we omit their discussion and refer readers to related literature [12, 16]. One lesser-known operator is perhaps **roll_right**, which pops the final trailing value, shifts all elements of the input array one component up, and inserts the first argument in place of the leading value. It ultimately returns the new array and

the original trailing element. This operation is useful for implementing exclusive scans from the inclusive **scan**.

We have omitted a handful of operations that are popular in second-order languages. First, we have no syntax for array indexing. This simplifies typing, as we need not worry about out-of-bounds access. More importantly, efficiently differentiating random access requires more detailed attention than we can pay it in this work [19]. Second, the language has no way of expressing in-place array updates or mutation. These operations implicitly sequentialize the program and hence can be handled using existing AD concepts, so we consider their omission acceptable. What remains, importantly, is a "parallel subset" of a representative second-order array language.

The language also newly introduces two built-in function transformations that invoke automatic differentiation:

$$\mathbf{jvp} : (a \to b) \to (a, a) \to (b, b)$$
$$\mathbf{vjp} : (a \to b) \to a \to (b, b \to a)$$

The transformation carried out by **jvp** is commonly known as forward-mode AD, while **vjp** corresponds to reverse-mode AD. The names are initialisms for the terms *Jacobian-vector product* and *vector-Jacobian product*, respectively. Section 3 discusses their semantics.

We also add a Zero t expression, representing a zero value of type t. We use this symbolic zero to implement AD of product types with sensible asymptotic complexity, namely avoiding the runtime and memory cost of unnecessarily constructing all-zero arrays. In a language that already contains optimized constructs for such replicated arrays (e.g. by implementing **broadcast** as a constant-time operation), they could replace this expression.

To make an expression Zero t well-defined, we have annotated array types with their sizes. Having omitted typing rules, the details of static size checking are also intentionally left out for brevity. Many systems implement some variant of this technique, whether with types containing statically known constants (e.g. the XLA compiler [1]), or with a more flexible approach based in dependent type theory [12, 19, 21]. Ultimately, this primarily a convenience for presentation. In a language that does not carry array sizes in types, the Zero constructor could be parameterized at run-time with integers denoting shape.

Our language is sufficiently simple that, in the context of AD, every type is isomorphic to its tangent types. We therefore do not reason about or denote tangent types separately.

3 Automatic Differentiation Rules

Automatic differentiation of in-language functions models differentiation of mathematical functions. Given a nice mathematical function $f : \mathbb{R}^n \to \mathbb{R}^m$, at a point $x \in \mathbb{R}^n$ its derivative $\partial f_x : \mathbb{R}^n \to \mathbb{R}^m$ is a linear map such that:

$$f(x + v) = f(x) + \partial f_x(v) + o(\|v\|), \quad \forall v \in \mathbb{R}^n . \tag{1}$$

Intuitively, ∂f_x encodes how a small perturbation v to x, the input to the original function f, pushes forward to a perturbation on the output $f(x)$. In the expression $\partial f_x(v)$, we sometimes refer to x as the *linearization point* for the derivative, and to v as the *input tangent*, *perturbation*, or *direction of differentiation*. Because ∂f_x is a linear function, we can define its transpose $\partial^\mathsf{T} f_x : \mathbb{R}^m \to \mathbb{R}^n$, itself a linear function, by:

$$\langle w, \partial f_x(v) \rangle = \langle \partial^\mathsf{T} f_x(w), v \rangle, \quad \forall v \in \mathbb{R}^n, w \in \mathbb{R}^m , \tag{2}$$

where $\langle \cdot, \cdot \rangle$ is the standard inner product on the appropriate space. In particular, when $m = 1$ so that $f : \mathbb{R}^n \to \mathbb{R}$, we can compute the gradient of f as $\nabla f(x) = \partial^\mathsf{T} f_x(1)$.[1] In the expression $\partial^\mathsf{T} f_x(w)$, we sometimes refer to w as the *output (co)tangent*.

With multiple arguments, like $g : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}^p$, we write the derivative at the point (x, y) as $\partial g_{x,y} : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}^p$. To differentiate with respect to just one argument, we write ∂_1 or ∂_2, where:

$$\partial g_{x,y}(v_1, v_2) = \partial_1 g_{x,y}(v_1) + \partial_2 g_{x,y}(v_2) . \tag{3}$$

In automatic differentiation, we often find it useful to evaluate a function together with its derivatives. Modeling that in a *compositional* way gives rise to what we call the JVP and VJP functions, which are the mathematical analogues of the **jvp** and **vjp** combinators in our programming language (Section 2). The *JVP function* of a function f is another function $\dot{\partial} f$ given by:

$$\dot{\partial} f(x, \dot{x}) = (f(x), \partial f_x(\dot{x})) . \tag{4}$$

This definition is compositional in the sense that $\dot{\partial}(f \circ g) = (\dot{\partial} f) \circ (\dot{\partial} g)$, by what is commonly called the chain rule in calculus. The *VJP function* of a function f is another function $\hat{\partial} f$ given by:

$$\hat{\partial} f(x) = (f(x), \hat{y} \mapsto \partial^\mathsf{T} f_x(\hat{y})) \tag{5}$$

and similarly $\hat{\partial}(f \circ g)$ can be written in terms of $\hat{\partial} f$ and $\hat{\partial} g$. This compositionality essentially enables us to focus only on differentiation rules for language primitives in AD.

While we are interested in performing AD on the entire language from Section 2, the rules specified here only differentiate a subset of it that can be obtained by a process called *normalization to first-order* or *defunctionalization* [13, 19, 20]. This normalization is common practice in array languages, in part because accelerators (such as GPUs) prefer code with few to no function indirections. In particular, all lambda expressions are expected to be inlined and beta-reduced prior to AD, and higher-order primitives such as **map** neither accept nor return arrays of functions. Similarly, we maintain that no AD-related primitives appear within a differentiated function. In circumstances where this might seem desirable (e.g. when taking higher-order derivatives), we can assume

[1] The gradient is defined as the unique vector $\nabla f(x) \in \mathbb{R}^n$ such that $\langle \nabla f(x), v \rangle = \partial f_x(v)$ for all vectors v.

that they have been reduced first (i.e. replaced by the result of the corresponding AD transformation).

For convenience of both presentation and implementation, we do allow our AD rules to emit unnormalized code themselves. To support this, one can interleave AD transformations with normalization to first-order, as recommended by Paszke et al. [19].

The interesting automatic differentiation rules are those concerning **map**, **reduce** and **scan**, covered in the subsequent section. All other language constructs have been considered in the past in other literature or projects, as they relate either to AD of scalar programs or first-order array operations.

3.1 Derivatives of Associative Functions

It is useful first to review the law of associativity and its implications for the derivatives of associative functions. Consider a monoid $(e, \odot)$, where we infer the set of monoid elements from the type of the neutral element e. The associative law for $\odot$ is:

$$(x \odot y) \odot z = x \odot (y \odot z).$$

Because this holds for any $x, y, z \in X$, it holds in particular in the neighborhood of any fixed x, y, z, implying that the equality also holds after differentiating both sides:

$$\partial\odot_{x\odot y,z}(\partial\odot_{x,y}(\dot{x},\dot{y}),\dot{z}) = \partial\odot_{x,y\odot z}(\dot{x},\partial\odot_{y,z}(\dot{y},\dot{z})). \quad (6)$$

Here, $\partial\odot_{x,y}(\dot{x},\dot{y})$ denotes the derivative of $\odot$ linearized at (x, y) and applied to $(\dot{x}, \dot{y})$.

Equation (6) is not an associative law for $\partial\odot$. In particular, $\partial\odot$ can be seen as a family of linear functions that depends on the linearization point, and each invocation in the above equation mentions a distinct member of this family. But Equation (6) does allow us to wrap the process of differentiation in an associative computation, so long as we take care to select the function $\partial\odot$ associated with the correct linearization points. Operations that obey laws similar to the one in Equation (6) are often said to be *pseudo-associative*.

In the following we assume that $\odot$ is reified in our language as a closed term of type $(a, a) \to a$. Similarly, $\partial\odot$ can also be reified as a closed term of type $(a, a) \to (a, a) \to a$, with the first pair corresponding to the linearization point, and the second arrow corresponding to a linear function.

3.2 Forward-Mode AD (jvp)

Forward-mode AD is the simpler of the two AD transformations. Differentiating most of our language's constructs, as well as its first-order array operations, is standard and needs no explanation. For completeness, all AD rules are given in Figure 2 (of Appendix A). We only discuss **map**, **scan** and **reduce**, with the latter two demanding the more careful treatment.

3.2.1 map.

Forward-mode differentiation of **map** is straightforward. Given an array x and its tangent $\dot{x}$, it suffices to **zip** them together and **map** the JVP of the original mapped function over such an array of pairs. This produces another array of pairs, which can be projected into two separate arrays by an **unzip** function defined as:

$$\textbf{unzip} = \lambda x : [n](a, b).\ [\textbf{map}(x, \text{fst}), \textbf{map}(x, \text{snd})]$$

3.2.2 reduce and scan.

Consider a monoid $M = (e, \odot)$ over elements of type t. Define a new structure

$$\partial M = ([e, \text{Zero } t], \oplus)$$

where $\oplus$, which has type $(a, a) \to (a, a) \to (a, a)$, is given by:

$$[x, \dot{x}] \oplus [y, \dot{y}] = [x \odot y,\ \partial\odot_{x,y}(\dot{x}, \dot{y})]\,.$$

Note that $\partial\odot$ corresponds to a nested invocation of **jvp** applied to the monoidal reduction.

Proposition 3.1. *The binary operation $\oplus$ is associative.*

Proof. Take any $[x, \dot{x}]$, $[y, \dot{y}]$ and $[z, \dot{z}]$ and consider two ways of associating the expression $[x, \dot{x}] \oplus [y, \dot{y}] \oplus [z, \dot{z}]$:

$$\begin{aligned}
&([x, \dot{x}] \oplus [y, \dot{y}]) \oplus [z, \dot{z}] \\
&\quad = [x \odot y,\ \partial\odot_{x,y}(\dot{x}, \dot{y})] \oplus [z, \dot{z}] \\
&\quad = [(x \odot y) \odot z,\ \partial\odot_{x\odot y,z}(\partial\odot_{x,y}(\dot{x}, \dot{y}), \dot{z})] \\
&\quad = [x \odot (y \odot z),\ \partial\odot_{x,y\odot z}(\dot{x}, \partial\odot_{y,z}(\dot{y}, \dot{z}))] \quad \text{(Equation (6))} \\
&\quad = [x, \dot{x}] \oplus [y \odot z, \partial\odot_{y,z}(\dot{y}, \dot{z})] \\
&\quad = [x, \dot{x}] \oplus ([y, \dot{y}] \oplus [z, \dot{z}])\,.
\end{aligned}$$

□

Proposition 3.2. *If e has type t and is a left (right) identity of $\odot$, then $[e, \text{Zero } t]$ is a left (right) identity of $\oplus$.*

Proof. Assume that e is a right identity of $\odot$. The proof for a left identity is symmetric.

Take an arbitrary $[x, \dot{x}]$ and consider the expression:

$$\begin{aligned}
[x, \dot{x}] \oplus [e, \text{Zero } t] &= [x \odot e, \partial\odot_{x,e}(\dot{x}, \text{Zero } t)] \\
&= [x, \partial\odot_{x,e}(\dot{x}, \text{Zero } t)]\,.
\end{aligned}$$

With Equation (3) we can simplify the second component to:

$$\partial\odot_{x,e}(\dot{x}, \text{Zero } t) = \partial_1\odot_{x,e}(\dot{x}) + \partial_2\odot_{x,e}(\text{Zero } t) = \partial_1\odot_{x,e}(\dot{x}).$$

Since $x' \odot e = x'$ for all x' (in particular x' neighboring x), differentiating both sides shows that $\partial_1\odot_{x,e}$ is an identity and so $\partial_1\odot_{x,e}(\dot{x}) = \dot{x}$. □

Having confirmed that ∂M is a (left) monoid, it remains to show that it can be used to push perturbations forward through our program.

Theorem 3.3. *The expression:*

$$\textbf{\textit{unzip}}(\textbf{\textit{scan}}_{[e,\text{Zero } t],\oplus}(\textbf{\textit{zip}}(a, \dot{a})))$$

computes the JVP of the expression $\textbf{\textit{scan}}_{e,\odot}(a)$, linearized at a, in the direction $\dot{a}$.

Proof. To fix simple semantics for scan without loss of generality, suppose that it performs prefix summation sequentially from left to right. What we prove under this assumption extends to the parallel setting (i.e. under other orderings) due to the associativity of $\oplus$.

Define an array of prefix sums x of size n via the procedure:

$$x[0] = a[0] \qquad x[i+1] = x[i] \odot a[i]$$

for $i = 0, \ldots, n-1$. Applying standard AD rules to this system of equations yields the following procedure for propagating forward sensitivities:

$$\dot{x}[0] = \dot{a}[0] \qquad \dot{x}[i+1] = \partial\odot_{x_i,a[i]}(\dot{x}_i, \dot{a}[i])$$

for $i = 0, \ldots, n-1$. We can combine both procedures into one by writing:

$$\begin{aligned}[x_0, \dot{x}_0] &= [a[0], \dot{a}[0]] \\ [x_{i+1}, \dot{x}_{i+1}] &= [x_i \odot a[i], \partial\odot_{x_i,a[i]}(\dot{x}_i, \dot{a}[i])] \\ &= [x_i, \dot{x}_i] \oplus [a[i], \dot{a}[i]]\end{aligned}$$

which is precisely the scan expressed in the claim. □

Since **reduce** can be written as a **scan** composed with a projection to select the final array element, the JVP of an expression $\mathbf{reduce}_{e,\odot}(a)$, linearized at a, in the direction $\dot{a}$ can be efficiently computed as:

$$\mathbf{reduce}_{[e,\text{Zero } t],\oplus}(\mathbf{zip}(a, \dot{a}))\,.$$

3.3 Reverse-Mode AD (vjp)

Although its definition is typically more involved, reverse-mode AD is often preferred to forward mode in practice, especially when a function's input dimension far exceeds its output dimension.

Similar to Equation (3), the monoid operation's derivative transpose $\partial^\mathsf{T}\odot$ decomposes into two components which we will denote by $\partial_1^\mathsf{T}\odot$ and $\partial_2^\mathsf{T}\odot$. When we use $\partial^\mathsf{T}\odot$ in a program, we expect that $\partial^\mathsf{T}\odot$ is reified as a function of type $(a, a) \to a \to (a, a)$. Then, $\partial_1^\mathsf{T}\odot$ and $\partial_2^\mathsf{T}\odot$ can be obtained by composing the term representing $\partial^\mathsf{T}\odot$ with left and right projection respectively.

The challenge is again in differentiating **map**, **reduce** and **scan**. VJPs for the rest of the language can be defined conventionally, as they are in many first-order languages.

3.3.1 map. Like its JVP, the VJP of **map** is relatively straightforward, but also presents an opportunity to underscore the role of normalization to first order, as mentioned in Section 2. Intuitively, we would like to replace the primal expression $\mathbf{map}(u, f)$ with $\mathbf{map}(u, \mathbf{vjp}\, f)$; this is roughly what the rule in Figure 3 does. Recall however that $\mathbf{vjp}\, f$ has type $a \to (b, b \to a)$, so that mapping it will result in an array of functions, and we would like to assume that **map** neither accepts nor returns arrays of functions. In other words, our rewrite rule generates unnormalized code.

If, hypothetically, one wanted to avoid producing unnormalized output, a useful observation is that the functions in such an array share the same code, and differ only in their closure ($\mathbf{vjp}\, f$ depends only on the body expression of f). Intuitively, we could replace each resulting function with its captured closure, and inline their common body expression at downstream sites of use. Still, we argue that this would needlessly conflate first-order normalization with AD. Instead, again, we recommend simply interleaving AD passes with normalization passes.

Finally, for simplicity the VJP rule for **map** assumes that the mapped function is closed. This is without loss of generality: otherwise the closure could be lambda-converted with free variables passed in as extra arguments through **broadcast**.

3.3.2 reduce and scan. Take $M = (e, \odot)$ to be an arbitrary left monoid over a type t. Consider carrying out $\mathbf{scan}_{e,\odot}(a)$ over an array a, sequentially from left to right. Abbreviate the prefix sums (which comprise the scan's output) as x_i and consider defining them by the procedure:

$$x[0] = a[0] \qquad x[i+1] = x[i] \odot a[i+1]\,. \tag{7}$$

for $i = 0, \ldots, n-1$. When computing the VJP, we must pull back the cotangents of each output (call it $\hat{c}_i$) to the input cotangents. Cotangent propagation rules can be obtained by applying VJP rules to the recurrence (7), which yields:

$$\hat{x}[n-1] = \hat{c}[n-1] \quad \hat{x}[i] = \partial_1^\mathsf{T}\odot_{x[i],a[i+1]}(\hat{x}[i+1]) + \hat{c}[i] \tag{8}$$

$$\hat{a}[0] = \hat{x}[0] \quad \hat{a}[i] = \partial_2^\mathsf{T}\odot_{x[i-1],a[i]}(\hat{x}[i])\,.$$

Here, $\hat{x}$ is the cotangent corresponding to x ($\hat{c}$ does not account for the influence of $x[i]$ on all $x[j]$ for $j > i$), and $\hat{a}$ is the cotangent corresponding to the input array. Since we are concerned with parallel evaluation of derivatives, the key challenge lies in solving the recursive definition of elements of $\hat{x}$. With an array of values $\hat{x}$ in hand, $\hat{a}$ can be computed by a single application of **map**.

Define a new binary operation $\otimes$ by:

$$[x_1, a_1, \hat{y}_1] \otimes [x_2, a_2, \hat{y}_2] \tag{9}$$

$$= [x_1, a_1 \odot a_2, \partial_1^\mathsf{T}\odot_{x_1,a_1}(\hat{y}_2) + \hat{y}_1]. \tag{10}$$

The following lemma will prove useful in establishing associativity of $\otimes$.

Lemma 3.4. *$\partial_1\odot$ satisfies:*

$$\partial_1\odot_{x\odot y,z} \circ \partial_1\odot_{x,y} = \partial_1\odot_{x,y\odot z}$$

Proof. Expand each application of $\partial\odot$ in Equation (6) into separate linear components that operate on one argument

each, as shown in Equation (3). For the left side of Equation (6):

$$\begin{aligned}
&\partial\odot_{x\odot y,z}(\partial\odot_{x,y}(\dot{x},\dot{z}),\dot{z})\\
&\quad= \partial_1\odot_{x\odot y,z}(\partial_1\odot_{x,y}(\dot{x}) + \partial_2\odot_{x,y}(\dot{y})) + \partial_2\odot_{x\odot y,z}(\dot{z})\\
&\quad= \partial_1\odot_{x\odot y,z}(\partial_1\odot_{x,y}(\dot{x})) + \partial_1\odot_{x\odot y,z}(\partial_2\odot_{x,y}(\dot{y}))\\
&\qquad\quad + \partial_2\odot_{x\odot y,z}(\dot{z})
\end{aligned}$$

and for the right side:

$$\begin{aligned}
&\partial\odot_{x,y\odot z}(\dot{x},\partial\odot_{y,z}(\dot{y},\dot{z}))\\
&\quad= \partial_1\odot_{x,y\odot z}(\dot{x}) + \partial_2\odot_{x,y\odot z}(\partial_1\odot_{y,z}(\dot{y}) + \partial_2\odot_{y,z}(\dot{z}))\\
&\quad= \partial_1\odot_{x,y\odot z}(\dot{x}) + \partial_2\odot_{x,y\odot z}(\partial_1\odot_{y,z}(\dot{y}))\\
&\qquad\quad + \partial_2\odot_{x,y\odot z}(\partial_2\odot_{y,z}(\dot{z}))\,.
\end{aligned}$$

Equation (6) shows that the two sides are equal as linear maps, and so the equalities shown here hold for all $\dot{x}$, $\dot{y}$ and $\dot{z}$. In particular, we could set $\dot{y}$ and $\dot{z}$ to Zero t values, which by linearity of $\partial_2\odot$ and $\partial_1\odot$ would eliminate their contributions to the output and yield:

$$\partial_1\odot_{x\odot y,z}(\partial_1\odot_{x,y}(\dot{x})) = \partial_1\odot_{x,y\odot z}(\dot{x})$$

proving the claim. □

Lemma 3.4 allows us to re-associate the applications of $\partial_1\odot$: arbitrarily long chains of $\partial_1\odot$ application can be collapsed into one simply by adjusting the linearization point.

The conclusion of Lemma 3.4 can also be transposed. Because linear composition distributes over transposition:

$$\begin{aligned}
\partial_1^\mathsf{T}\odot_{x,y}\circ\partial_1^\mathsf{T}\odot_{x\odot y,z} &= (\partial_1\odot_{x\odot y,z}\circ\partial_1\odot_{x,y})^\mathsf{T}\\
&= (\partial_1\odot_{x,y\odot z})^\mathsf{T}\\
&= \partial_1^\mathsf{T}\odot_{x,y\odot z}
\end{aligned}$$

Equipped with these observations, we establish that $\otimes$ defines a semigroup with a suitable right identity (which we will call a *right monoid*).

Proposition 3.5. *The operation $\otimes$ is associative.*

Proof.

$$\begin{aligned}
&[x_1,a_1,\hat{y}_1]\otimes([x_2,a_2,\hat{y}_2]\otimes[x_3,a_3,\hat{y}_3])\\
&= [x_1,a_1,\hat{y}_1]\otimes[x_2,a_2\odot a_3,\hat{y}_2+\partial_1^\mathsf{T}\odot_{x_2,a_2}(\hat{y}_3)]\\
&= [x_1,a_1\odot a_2\odot a_3,\hat{y}_1+\partial_1^\mathsf{T}\odot_{x_1,a_1}(\hat{y}_2+\partial_1^\mathsf{T}\odot_{x_2,a_2}(\hat{y}_3))]\\
&= [x_1,a_1\odot a_2\odot a_3,\hat{y}_1+\partial_1^\mathsf{T}\odot_{x_1,a_1}(\hat{y}_2)\\
&\qquad\qquad + \partial_1^\mathsf{T}\odot_{x_1,a_1}(\partial_1^\mathsf{T}\odot_{x_2,a_2}(\hat{y}_3))]\\
&= [x_1,a_1\odot a_2\odot a_3,\hat{y}_1+\partial_1^\mathsf{T}\odot_{x_1,a_1}(\hat{y}_2)+\partial_1^\mathsf{T}\odot_{x_1,a_1\odot a_2}(\hat{y}_3)]\\
&= [x_1,a_1\odot a_2,\hat{y}_1+\partial_1^\mathsf{T}\odot_{x_1,a_1}(\hat{y}_2)]\otimes[x_3,a_3,\hat{y}_3]\\
&= ([x_1,a_1,\hat{y}_1]\otimes[x_2,a_2,\hat{y}_2])\otimes[x_3,a_3,\hat{y}_3]
\end{aligned}$$

□

Proposition 3.6. *The value $[e,e,\text{Zero } t]$ is a right-identity of $\otimes$.*

Proof. Recall that $\partial_1^\mathsf{T}\odot_{x,a}(\text{Zero } t) = \text{Zero } t$, by the linearity of $\partial_1^\mathsf{T}\odot$. We then have:

$$\begin{aligned}
[x,a,\hat{y}]\otimes[e,e,\text{Zero } t] &= [x,a\odot e,\partial_1^\mathsf{T}\odot_{x,a}(\text{Zero } t)+\hat{y}]\\
&= [x,a,\hat{y}]
\end{aligned}$$

□

Theorem 3.7. *Let m be an array of size n given by:*

$$m = \textbf{\textit{zip}}(x, snd(\textbf{\textit{roll\_left}}(a,e)), \hat{c})$$

where ***roll_left*** *is defined as:*

$$\begin{aligned}
&\lambda a:[n]t.\ \lambda v:t.\\
&\quad \textit{let } (x',v') = \textbf{\textit{roll_right}}(v, \textbf{\textit{reverse}}(x))\\
&\quad \textit{in } (v', \textbf{\textit{reverse}}(x'))\,.
\end{aligned}$$

Also define ***scan_right*** *as:*

$$\begin{aligned}
&\lambda(e,\odot):\mathit{RightMonoid}\ t.\ \lambda a:[n]t.\\
&\quad \textbf{\textit{reverse}}(\textbf{\textit{scan}}_{e,(\mathit{flip}\ \odot)}(\textbf{\textit{reverse}}(a))).
\end{aligned}$$

Note that due to the flip operator, ***scan_right*** *expects e to be the right identity of $\odot$, as emphasized in the RightMonoid t argument type. Then the projection onto the third component of the expression:*

$$\textbf{\textit{scan\_right}}_{[e,e,\mathit{Zero}\ t],\otimes}(m)$$

yields the array of cotangent values $\hat{x}_i$.

Proof. As in Theorem 3.3, we fix a right-to-left sequential evaluation of the scan over $\otimes$:

$$\begin{aligned}
[x'_{n-1}, r_{n-1}, \hat{x}_{n-1}] &= [x[n-1], e, \hat{c}[n-1]]\\
[x'_i, r_i, \hat{x}_i] &= [x[i], a[i+1], \hat{c}[i]]\otimes[x'_{i+1}, r_{i+1}, \hat{x}_{i+1}]\\
&= [x[i], a[i+1]\odot r_{i+1},\\
&\qquad \partial_1^\mathsf{T}\odot_{x[i],a[i+1]}(\hat{x}_{i+1})+\hat{c}[i]]
\end{aligned}$$

By an easy inductive argument, we do have that $\hat{x}_{n-1} = \hat{x}[n-1]$ (as defined above), and given that $\hat{x}_{i+1} = \hat{x}[i+1]$, the formula used to compute $\hat{x}_i$ matches exactly the recursive definition of $\hat{x}[i]$. □

The array $\hat{a}$ is *almost* a **map** of $\hat{x}$, except for one detail: the first element is defined differently from the rest. Fortunately, by arguing analogously to Proposition 3.2, one can show that $\partial_2^\mathsf{T}\odot_{e,v}(\dot{v}) = v$. We can use this property to shift the left-identity into the first element of the x array, rendering $\hat{a}$ a uniform **map** of the result:

$$x_{\text{shift}} = \text{fst}(\textbf{roll\_right}(x,e)) \tag{11}$$

$$\hat{a} = \textbf{map}(\textbf{zip}(\textbf{zip}(x_{\text{shift}}, a), \hat{x}), \partial_2^\mathsf{T}\odot) \tag{12}$$

Reverse-mode differentiation of **reduce** proceeds similarly: only $\hat{c}[n]$ is non-zero, and the remaining elements of $\hat{c}$ can be eliminated from the equation defining $\hat{x}[i]$.

Admittedly, the *left monoid* constraint is not a standard one to place on the algebraic structure parameterizing **scan**. The two common choices are either a monoid or a semigroup.

In the following we briefly discuss the rationale for this choice, and alternative approaches that make this approach adaptable to other languages.

Left monoid vs. monoid. The VJP described by Theorem 3.7 does not preserve monoidal laws across differentiation: the $\otimes$ operator, over which the VJP must **scan**, has only a one-sided identity (because of how it operates on the first component of its arguments). The derivative rule, as is, is therefore incompatible with a **scan** language construct that requires a regular monoid. The resulting algebraic structure is nonetheless a semigroup. In a language that supports sum types, any semigroup can be converted to a monoid by extending the type with a special identity element.

Left monoid vs. semigroup. The VJP described by Theorem 3.7 could be adapted for a **scan** with a semigroup constraint, but its presentation would be more involved. The two places where we use the left identity of the input monoid are: (i) in the **roll_left** before the **scan_right**, and (ii) in the **roll_right** that makes the definition of $\hat{a}$ expressible as a **map**. The element that we insert in (ii) could be replaced by any other well-typed value without affecting the cotangents $\hat{x}$ computed. However, a left identity is needed in order for $\partial_2^{\mathsf{T}}\odot$ to have the identity property that we rely on in Equation (12).

4 Efficiency

Efficiency is central to automatic differentiation. To define efficiency, we can use a cost model: for each closed term e, let $\$[\![e]\!]$ be a nonnegative integer representing the number of FLOPs to evaluate e. We can use a cost model that closely follows Bernstein et al. [4], so we do not specify its complete rules here.

With a cost model in place, the ideal AD efficiency statement is, in words, that evaluating a derivative costs only a small constant multiple of the cost of evaluating the primal function, regardless of the function. The following gives such a statement for JVPs.

Proposition 4.1. *There exists a positive constant α such that for all fully-reduced terms u, v and all well-typed first-order functions f we have,*

$$\$[\![(\boldsymbol{jvp}\ f)(u,v)]\!] \leq (1+\alpha)\cdot \$[\![f(u)]\!].$$

This claim holds true for the JVP rules given here. A proof would analyze each JVP rule and compare the cost of JVP evaluation to the cost of the primal evaluation alone, exactly analogous to the argument in Bernstein et al. [4].

Unfortunately the analogous claim for VJPs does not hold true for **scan** given the rule presented here. Notice that the VJP rule for scan involves recomputing primal up-sweep values. This recomputation results in overhead factors that can increase with the level of nesting of scans, rather than being independent of the function to be differentiated.

To see the issue, consider an example of nesting scans to compute cumulative products of cumulative products (of cumulative products, etc.). Concretely, using n and k as meta-variables, we define a sequence of functions recursively via:

$$f_1 = \lambda a : [n]\mathsf{Float}\ .\ \mathbf{scan}_{1,\times}(a),$$
$$f_k = \lambda a : [n]^k\mathsf{Float}\ .\ \mathbf{scan}_{1_{k-1},\odot_{k-1}}(a),$$

for $k = 2, 3, \ldots$, where we define the associative binary operator $\odot_k$ as:

$$\odot_k = \lambda a{:}\,[n]^k\mathsf{Float}.\ \lambda b{:}\,[n]^k\mathsf{Float}.\ \mathbf{map}(\mathbf{zip}(f_k(a), f_k(b)), \times_{k-1})$$

for $k = 1, 2, \ldots$, and where we define $\times_k$ and 1_k as

$$\times_0 = \times,$$
$$\times_k = \lambda a{:}\,[n]^k\mathsf{Float}.\ \lambda b{:}\,[n]^k\mathsf{Float}.\ \mathbf{map}(\mathbf{zip}(a,b), \times_{k-1}),$$
$$1_0 = 1, \quad 1_k = \mathbf{broadcast}(1_{k-1}), \quad k = 1, 2, \ldots.$$

For this example, denote by c_k the cost of evaluating the VJP of f_k,

$$c_k = \$[\![\mathsf{snd}\ (\mathbf{vjp}\ f_k\ u_k)\ v_k]\!],$$

for appropriate fully-evaluated terms u_k and v_k. Observe that because the definition of $\otimes$ in (9) includes an application of the primal monoid operation $\odot$, and because evaluating the VJP of f_k requires at least n evaluations of the VJPs of f_{k-1} from the recursion (with no work sharing), we have

$$c_k \geq \$[\![f_k(u_k)]\!] + n * c_{k-1}.$$

Furthermore, observe $\$[\![f_k(u_k)]\!] \geq n^k$, and so we have

$$c_k \geq n^k + n\cdot n^{k-1} + n^2\cdot n^{k-2} + \cdots + n^k = kn^k.$$

Thus the recomputation costs alone incur a multiplicative cost increase that scales with k, contra the ideal automatic differentiation efficiency which would be independent of k.

This nesting efficiency issue is a significant limitation of using **scan** as a primitive for automatic differentiation. However, even with this VJP rule, compiler optimizations that are downstream of AD, such as common-subexpression elimination, could combat the inefficiency. Further, this kind of recomputation can be desirable on hardware accelerators, where FLOPs are cheap and memory is expensive. But to have the option of avoiding this redundant recomputation, it may be necessary to replace **scan** with other language primitives.

5 Conclusion and Future Work

We have presented a framework for performing forward- and reverse-mode automatic differentiation of standard second-order array programming languages. Our method generalizes the current state-of-the-art AD systems, which would either be too limited (e.g. to only first-order array programs), or would force a sequentialization of the derivative program,

losing parallelism—arguably one of the central features of array programming languages.

However, our method has efficiency drawbacks that depart from standard expectations set in the context of scalar and first-order array programs, essentially arising from the **scan** primitive giving up control over re-association to an underlying compiler. We conjecture (from preliminary progress) that basing the same language in a different set of related primitives would allow us to alleviate this problem. Another reason for changing primitives is that the standard definition of **scan** presents a challenge in decomposing the VJP into linearization and transposition, as discussed in Frostig et al. [11]. Conforming to such a decomposition eases implementation and rule-writing dramatically.

Another potential avenue to explore is to exploit invertibility in monoidal functions, e.g. in multiplication of non-zero numbers (division being the inverse).

A Forward-Mode AD Rules

Figure 2 presents a complete set of rules for forward-mode AD of our second-order language (Figure 1). There are two collections of rewrites that determine the process.

The first of them:

$$\Delta \vdash \dot{\partial}\mathsf{F}[f] \rightsquigarrow \dot{\partial} f\,,$$

which carries an environment that maps primal variables (i.e. those that appear in the original program) to variables denoting their tangent values, rewrites a primal function f into its JVP function $\dot{\partial} f$. Every instance of the expression **jvp** f is replaced by $\dot{\partial} f$ as given by

$$\emptyset \vdash \dot{\partial}\mathsf{F}[f] \rightsquigarrow \dot{\partial} f\,.$$

The second rewrite $\Delta \vdash u \rightsquigarrow [u', \dot{u}]$ similarly uses the primal-tangent mapping Δ to compute the JVP of an expression u. If u is of type t, then it is expected to return an expression of type (t, t), with the primal quantity as its first component equal and the corresponding tangent as its second component.

B Reverse-Mode AD Rules

In Figure 3 we list a complete set of rewrites that implements reverse-mode AD in our second-order language. Similarly to forward-mode, the process is controlled by two types of rewrite rules.

The first one is used to implement **vjp** itself, because given

$$\hat{\partial}\mathsf{F}[f] \rightsquigarrow \hat{\partial} f$$

each instance of **vjp** f can be replaced by $\hat{\partial} f$.

The second rewrite

$$\Delta, \hat{t}, e \rightsquigarrow E_p, E_t, e', \Delta'$$

is more involved. It takes as an input an environment Δ mapping primal variables to variables denoting their cotangents, an expression e and an expression $\hat{t}$ denoting the cotangent of the result of e. The task of this rewrite is to use this information to update the cotangent values of all free variables occurring in e, and to update Δ with that information, yielding Δ'. But, it has three more outputs that enable this process. First, E_p is a *context* (a sequence of let-bindings terminated with a hole $\bullet$) which makes it possible to break up the whole expression into smaller intermediates, some of which can be shared by the cotangent computation. The expectation is that all variables bound here are considered in-scope for all other outputs. Second, E_t is another context, but this time it corresponds to the cotangent computation we generate. Any variable bound in E_t is only visible in Δ'. Finally, e' is an expression that computes a value equal to e, but e.g. can use the intermediates bound in E_p for efficiency.

$$\boxed{\Delta \vdash \dot{\partial}\mathsf{F}[f] \rightsquigarrow \dot{\partial} f}$$

$$\frac{\Delta, x \to t \vdash e \rightsquigarrow [e', \dot{e}]}{\Delta \vdash \dot{\partial}\mathsf{F}[\lambda x\!:\!t.\ e] \rightsquigarrow \lambda x\!:\!t.\ \lambda \dot{x}\!:\!t.\ [e', \dot{e}]} \qquad \frac{\Delta \vdash \dot{\partial}\mathsf{F}[\lambda x\!:\!t.\ f(x)] \rightsquigarrow \dot{\partial} f}{\Delta \vdash \dot{\partial}\mathsf{F}[f] \rightsquigarrow \dot{\partial} f}$$

$$\boxed{\Delta \vdash e \rightsquigarrow [e', \dot{e}]}$$

$$\frac{\Delta = \ldots, x \to \dot{x}, \ldots}{\Delta \vdash x \rightsquigarrow [x, \dot{x}]}\ \textsc{VarTan} \qquad \frac{x \notin \Delta \qquad x : t}{\Delta \vdash x \rightsquigarrow [x, \mathsf{Zero}\ t]}\ \textsc{VarZero}$$

$$\Delta \vdash \mathsf{Zero}\ t \rightsquigarrow [\mathsf{Zero}\ t, \mathsf{Zero}\ t]\ \textsc{Zero}$$

$$\frac{l : t}{\Delta \vdash l \rightsquigarrow [l, \mathsf{Zero}\ t]}\ \textsc{Lit} \qquad \frac{\Delta \vdash u \rightsquigarrow [u', \dot{u}] \qquad \Delta \vdash w \rightsquigarrow [w', \dot{w}]}{\Delta \vdash [u, w] \rightsquigarrow [[u', w'], [\dot{u}, \dot{w}]]}\ \textsc{PairCon}$$

$$\frac{\Delta \vdash u \rightsquigarrow [u', \dot{u}]}{\Delta \vdash \mathsf{fst}(u) \rightsquigarrow [\mathsf{fst}(u'), \mathsf{fst}(\dot{u})]}\ \textsc{PairLeft} \qquad \frac{\Delta \vdash u \rightsquigarrow [u', \dot{u}]}{\Delta \vdash \mathsf{snd}(u) \rightsquigarrow [\mathsf{snd}(u'), \mathsf{snd}(\dot{u})]}\ \textsc{PairRight}$$

$$\frac{\Delta \vdash u \rightsquigarrow \dot{u}' \qquad \Delta, x \to \dot{x} \vdash w \rightsquigarrow \dot{w}'}{\Delta \vdash \mathsf{let}\ x\!:\!t = u\ \mathsf{in}\ w \rightsquigarrow \mathsf{let}\ [x, \dot{x}]\!:\!(t, t) = \dot{u}'\ \mathsf{in}\ \dot{w}'}\ \textsc{Let}$$

$$\frac{\Delta \vdash u \rightsquigarrow [u', \dot{u}] \qquad \Delta \vdash \dot{\partial}\mathsf{F}[f] \rightsquigarrow \dot{\partial} f}{\Delta \vdash \mathbf{map}(u, f) \rightsquigarrow \mathbf{unzip}(\mathbf{map}(\mathbf{zip}(\mathsf{fst}(\dot{u}), \mathsf{snd}(\dot{u})), \dot{\partial} f)}\ \textsc{Map}$$

$$\frac{e : t \qquad \Delta \vdash u \rightsquigarrow [u', \dot{u}] \qquad a \to \dot{a}, b \to \dot{b} \vdash \odot(a, b) \rightsquigarrow [r, \dot{r}] \qquad \oplus = \lambda[[a, \dot{a}], [b, \dot{b}]]\!:\!((t, t), (t, t)).\ [r, \dot{r}]}{\Delta \vdash \mathbf{reduce}([e, \odot], u) \rightsquigarrow \mathbf{unzip}(\mathbf{reduce}([[e, \mathsf{Zero}\ t], \oplus], \mathbf{zip}(u', \dot{u})))}\ \textsc{Reduce}$$

$$\frac{e : t \qquad \Delta \vdash u \rightsquigarrow [u', \dot{u}] \qquad a \to \dot{a}, b \to \dot{b} \vdash \odot(a, b) \rightsquigarrow [r, \dot{r}] \qquad \oplus = \lambda[[a, \dot{a}], [b, \dot{b}]]\!:\!((t, t), (t, t)).\ [r, \dot{r}]}{\Delta \vdash \mathbf{scan}([e, \odot], u) \rightsquigarrow \mathbf{unzip}(\mathbf{scan}([[e, \mathsf{Zero}\ t], \oplus], \mathbf{zip}(u', \dot{u})))}\ \textsc{Scan}$$

$$\frac{\Delta \vdash u \rightsquigarrow [u', \dot{u}] \qquad \Delta \vdash w \rightsquigarrow [w', \dot{w}]}{\Delta \vdash \mathbf{zip}(u, w) \rightsquigarrow [\mathbf{zip}(u', w'), \mathbf{zip}(\dot{u}, \dot{w})]}\ \textsc{Zip}$$

$$\frac{\Delta \vdash u \rightsquigarrow [u', \dot{u}]}{\Delta \vdash \mathbf{reverse}(u) \rightsquigarrow [\mathbf{reverse}(u'), \mathbf{reverse}(\dot{u})]}\ \textsc{Reverse}$$

$$\frac{\Delta \vdash u \rightsquigarrow [u', \dot{u}] \qquad \Delta \vdash w \rightsquigarrow [w', \dot{w}]}{\Delta \vdash \mathbf{roll\_right}(u, w) \rightsquigarrow [\mathbf{roll\_right}(u', w'), \mathbf{roll\_right}(\dot{u}, \dot{w})]}\ \textsc{Roll}$$

$$\frac{\Delta \vdash u \rightsquigarrow [u', \dot{u}]}{\Delta \vdash \mathbf{transpose}(u) \rightsquigarrow [\mathbf{transpose}(u'), \mathbf{transpose}(\dot{u})]}\ \textsc{Transpose}$$

Figure 2. Forward-mode differentiation rules. Note that for simplicity of presentation we assume that the output of the main rewrite is a pair constructor that can be unpacked into a primal and tangent component. In a real system this might require an introduction of new let bindings (e.g. to project out the components of the expression originating from the Let rule). See Appendix A for a longer description of the transforms above.

$$\boxed{\hat{\partial}\mathsf{F}[f] \rightsquigarrow \hat{\partial} f}$$

$$\frac{\emptyset, \hat{t}, e \rightsquigarrow E_p, E_t, e', \Delta}{\hat{\partial}\mathsf{F}[\lambda x\!:\!t.\ e] \rightsquigarrow \lambda x\!:\!t.\ E_p[(e', \lambda \hat{x}\!:\!t.\ E_t[\Delta[x]])]} \qquad \frac{\hat{\partial}\mathsf{F}[\lambda x.\ f(x)] \rightsquigarrow \hat{\partial} f}{\hat{\partial}\mathsf{F}[f] \rightsquigarrow \hat{\partial} f}$$

$$\boxed{\Delta, \hat{t}, e \rightsquigarrow E_p, E_t, e', \Delta'}$$

$$\Delta, \hat{x}, x \rightsquigarrow \bullet, \text{let } \hat{x}'\!:\!t = \hat{x} + \Delta[x] \text{ in } \bullet, x, \Delta[x \to \hat{x}'] \ \textsc{Var} \qquad \Delta, \hat{t}, l \rightsquigarrow \bullet, \bullet, l, \Delta \ \textsc{Lit} \qquad \Delta, \hat{t}, \text{Zero } t \rightsquigarrow \bullet, \bullet, \text{Zero } t, \Delta \ \textsc{Zero}$$

$$\frac{\Delta, \mathsf{fst}(\hat{t}), u \rightsquigarrow E_p^1, E_t^1, u', \Delta' \qquad \Delta', \mathsf{snd}(\hat{t}), w \rightsquigarrow E_p^2, E_t^2, w', \Delta''}{\Delta, \hat{t}, [u, w] \rightsquigarrow E_p^1 \circ E_p^2, E_t^1 \circ E_t^2, [u', w'], \Delta''} \ \textsc{PairCon}$$

$$\frac{\Delta, [\hat{u}, \text{Zero } t], u \rightsquigarrow E_p, E_t, u', \Delta'}{\Delta, \hat{u}, \mathsf{fst}(u) \rightsquigarrow E_p, E_t, \mathsf{fst}(u'), \Delta'} \ \textsc{PairLeft} \qquad \frac{\Delta, [\text{Zero } t, \hat{u}], u \rightsquigarrow E_p, E_t, u', \Delta'}{\Delta, \hat{u}, \mathsf{snd}(u) \rightsquigarrow E_p, E_t, \mathsf{snd}(u'), \Delta'} \ \textsc{PairRight}$$

$$\frac{\Delta, \hat{w}, w \rightsquigarrow E_p^w, E_t^w, w', \Delta' \qquad \Delta' \setminus x, \Delta'[x], u \rightsquigarrow E_p^u, E_t^u, u', \Delta''}{\Delta, \hat{w}, \text{let } x\!:\!t = u \text{ in } w \rightsquigarrow E_p^u \circ \text{let } x\!:\!t = u' \text{ in } E_p^w, E_t^w \circ E_t^u, w', \Delta''} \ \textsc{Let}$$

$$\frac{\mathsf{FV}(f) = \emptyset \qquad \hat{\partial}\mathsf{F}[f] \rightsquigarrow \hat{\partial} f \qquad \Delta, \mathbf{map}(\mathbf{zip}(\mathsf{snd}(x), \hat{t}), \lambda y.\ \mathsf{fst}(y)(\mathsf{snd}(y))), u \rightsquigarrow E_p, E_t, u', \Delta'}{\Delta, \hat{t}, \mathbf{map}(u, f) \rightsquigarrow E_p \circ (\text{let } x = \mathbf{map}(u', \hat{\partial} f) \text{ in } \bullet), E_t, \mathsf{fst}(x), \Delta'} \ \textsc{Map}$$

$$E_t = \begin{cases} \text{let } \hat{y}' = \mathsf{snd}(\mathbf{roll\_left}(\mathbf{broadcast}(\text{Zero } t), \hat{y})) \text{ in} \\ \text{let } m = \mathbf{zip}(x, \mathsf{snd}(\mathbf{roll\_left}(u, e), \hat{y}')) \text{ in} \\ \text{let } \hat{x} = \mathbf{scan\_right}([[e, e, \text{Zero }], \otimes], m) \text{ in} \\ \text{let } x_{\text{shift}} = \mathsf{fst}(\mathbf{roll\_right}(x, e)) \text{ in} \\ \text{let } \hat{a} = \mathbf{map}(\mathbf{zip}(\mathbf{zip}(x_{\text{shift}}, a), \hat{x}), \partial_2^{\mathsf{T}}\odot) \text{ in } \bullet \end{cases}$$

$$\frac{\otimes \text{ and } \partial_2^{\mathsf{T}}\odot \text{ as defined in the text} \qquad \hat{y} : t \qquad \Delta, \hat{a}, u \rightsquigarrow E_p^u, E_t^u, u', \Delta'}{\Delta, \hat{y}, \mathbf{reduce}([e, \odot], u) \rightsquigarrow E_p^u \circ (\text{let } x = \mathbf{scan}([e, \odot], u') \text{ in } \bullet), E_t \circ E_t^u, \mathbf{reduce}([e, \odot], u'), \Delta'} \ \textsc{Reduce}$$

$$E_t = \begin{cases} \text{let } m = \mathbf{zip}(x, \mathsf{snd}(\mathbf{roll\_left}(u, e), \hat{y}')) \text{ in} \\ \text{let } \hat{x} = \mathbf{scan\_right}([[e, e, \text{Zero }], \otimes], m) \text{ in} \\ \text{let } x_{\text{shift}} = \mathsf{fst}(\mathbf{roll\_right}(x, e)) \text{ in} \\ \text{let } \hat{a} = \mathbf{map}(\mathbf{zip}(\mathbf{zip}(x_{\text{shift}}, a), \hat{x}), \partial_2^{\mathsf{T}}\odot) \text{ in } \bullet \end{cases}$$

$$\frac{\otimes \text{ and } \partial_2^{\mathsf{T}}\odot \text{ as defined in the text} \qquad \Delta, \hat{a}, u \rightsquigarrow E_p^u, E_t^u, u', \Delta'}{\Delta, \hat{t}, \mathbf{scan}([e, \odot], u) \rightsquigarrow E_p^u \circ (\text{let } x = \mathbf{scan}([e, \odot], u) \text{ in } \bullet), E_t \circ E_t^u, x, \Delta'} \ \textsc{Scan}$$

$$\frac{\Delta, \mathbf{map}(\hat{t}, \mathsf{fst}), u \rightsquigarrow E_p^u, E_t^u, u', \Delta' \qquad \Delta', \mathbf{map}(\hat{t}, \mathsf{snd}), w \rightsquigarrow E_p^w, E_t^w, w', \Delta''}{\Delta, \hat{t}, \mathbf{zip}(u, w) \rightsquigarrow E_p^u \circ E_p^w, E_t^w \circ E_t^u, \mathbf{zip}(u', w'), \Delta''} \ \textsc{Zip}$$

$$\frac{\Delta, \mathbf{reverse}(\hat{t}), u \rightsquigarrow E_p, E_t, u', \Delta'}{\Delta, \hat{t}, \mathbf{reverse}(u) \rightsquigarrow E_p, E_t, \mathbf{reverse}(u'), \Delta'} \ \textsc{Reverse} \qquad \frac{\Delta, \mathbf{transpose}(\hat{t}), u \rightsquigarrow E_p, E_t, u', \Delta'}{\Delta, \hat{t}, \mathbf{transpose}(u) \rightsquigarrow E_p, E_t, \mathbf{transpose}(u'), \Delta'} \ \textsc{Transpose}$$

$$\frac{\Delta, \hat{u}, u \rightsquigarrow E_p^u, E_t^u, u', \Delta' \qquad \Delta', \hat{w}, w \rightsquigarrow E_p^w, E_t^w, w', \Delta''}{\Delta, \hat{t}, \mathbf{roll\_right}(u, w) \rightsquigarrow E_p^u \circ E_p^w, \text{let } (\hat{w}, \hat{u}) = \mathbf{roll\_left}(\mathsf{fst}(\hat{t}), \mathsf{snd}(\hat{t})) \text{ in } E_t^w \circ E_t^u, \mathbf{roll\_right}(u', w'), \Delta''} \ \textsc{Roll}$$

Figure 3. Reverse-mode differentiation rules. We assume that there is no shadowing in the input program. Any variables generated by the rules are assumed to be completely fresh as well. See Appendix B for a longer description.

References

[1] [n.d.]. XLA: Optimizing Compiler for Machine Learning. https://www.tensorflow.org/xla. Online; accessed 1 May 2021.

[2] 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. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)*. USENIX Association, USA, 265–283.

[3] Martín Abadi and Gordon D. Plotkin. 2019. A Simple Differentiable Programming Language. *Proc. ACM Program. Lang.* 4, POPL, Article 38 (Dec. 2019), 28 pages. https://doi.org/10.1145/3371106

[4] Gilbert Bernstein, Michael Mara, Tzu-Mao Li, Dougal Maclaurin, and Jonathan Ragan-Kelley. 2020. Differentiating A Tensor Language. *arXiv preprint arXiv:2008.11256* (2020).

[5] Guy E. Blelloch. 1993. Prefix Sums and Their Applications. In *Synthesis of Parallel Algorithms*, John H. Reif (Ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, Chapter 1, 35–60.

[6] Guy E. Blelloch and Bruce M. Maggs. 2010. *Parallel Algorithms* (2 ed.). Chapman & Hall/CRC, 25.

[7] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. 2018. *JAX: composable transformations of Python+NumPy programs.* http://github.com/google/jax

[8] Aloïs Brunel, Damiano Mazza, and Michele Pagani. 2019. Backpropagation in the Simply Typed Lambda-Calculus with Linear Negation. *Proc. ACM Program. Lang.* 4, POPL, Article 64 (Dec. 2019), 27 pages. https://doi.org/10.1145/3371132

[9] Conal Elliott. 2018. The Simple Essence of Automatic Differentiation. *Proc. ACM Program. Lang.* 2, ICFP, Article 70 (July 2018), 29 pages. https://doi.org/10.1145/3236765

[10] Roy Frostig, Matthew Johnson, and Chris Leary. 2018. Compiling machine learning programs via high-level tracing. *Machine Learning and Systems (MLSys).* https://mlsys.org/Conferences/doc/2018/146.pdf

[11] Roy Frostig, Matthew J. Johnson, Dougal Maclaurin, Adam Paszke, and Alexey Radul. 2021. Decomposing reverse-mode automatic differentiation. In *LAFI 2021 workshop at POPL.* https://arxiv.org/abs/2105.09469

[12] Troels Henriksen, Niels G. W. Serup, Martin Elsman, Fritz Henglein, and Cosmin E. Oancea. 2017. Futhark: Purely Functional GPU-Programming with Nested Parallelism and in-Place Array Updates. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017).* Association for Computing Machinery, New York, NY, USA, 556–571. https://doi.org/10.1145/3062341.3062354

[13] Anders Kiel Hovgaard, Troels Henriksen, and Martin Elsman. 2019. High-Performance Defunctionalisation in Futhark. In *Trends in Functional Programming*, Michał Pałka and Magnus Myreen (Eds.). Springer International Publishing, Cham, 136–156. https://doi.org/10.1007/978-3-030-18506-0_7

[14] P. Hovland and C. Bischof. 1998. Automatic Differentiation for Message-Passing Parallel Programs. In *Proceedings of the 12th. International Parallel Processing Symposium on International Parallel Processing Symposium (IPPS '98).* IEEE Computer Society, USA, 98–104. https://doi.org/10.1109/IPPS.1998.669896

[15] Dougal Maclaurin, Alexey Radul, Matthew J. Johnson, and Dimitrios Vytiniotis. 2019. Dex: array programming with typed indices. *NeurIPS workshop: Program Transformations for Machine Learning* (2019).

[16] Kiminori Matsuzaki and Kento Emoto. 2010. Implementing Fusion-Equipped Parallel Skeletons by Expression Templates. In *Implementation and Application of Functional Languages*, Marco T. Morazán and Sven-Bodo Scholz (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 72–89. https://doi.org/10.1007/978-3-642-16478-1_5

[17] Damiano Mazza and Michele Pagani. 2021. Automatic Differentiation in PCF. *Proc. ACM Program. Lang.* 5, POPL, Article 28 (Jan. 2021), 27 pages. https://doi.org/10.1145/3434309

[18] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.). Curran Associates, Inc., 8024–8035. http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[19] Adam Paszke, Daniel Johnson, David Duvenaud, Dimitrios Vytiniotis, Alexey Radul, Matthew Johnson, Jonathan Ragan-Kelley, and Dougal Maclaurin. 2021. Getting to the Point: Index Sets and Parallelism-Preserving Autodiff for Pointful Array Programming. *Proc. ACM Program. Lang.* 5, ICFP, Article 88 (Aug. 2021). https://doi.org/10.1145/3473593

[20] John C. Reynolds. 1972. Definitional Interpreters for Higher-Order Programming Languages. In *Proceedings of the ACM Annual Conference - Volume 2 (ACM '72).* Association for Computing Machinery, New York, NY, USA, 717–740. https://doi.org/10.1145/800194.805852

[21] Justin Slepak, Olin Shivers, and Panagiotis Manolios. 2014. An Array-Oriented Language with Static Rank Polymorphism. In *Programming Languages and Systems*, Zhong Shao (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 27–46. https://doi.org/10.1007/978-3-642-54833-8_3

[22] Dimitrios Vytiniotis, Dan Belov, Richard Wei, Gordon Plotkin, and Martin Abadi. 2019. The differentiable curry. In *NeurIPS 2019 Workshop Program Transformations.* https://openreview.net/forum?id=ryxuz9SzDB

[23] Fei Wang, Daniel Zheng, James Decker, Xilun Wu, Grégory M. Essertel, and Tiark Rompf. 2019. Demystifying Differentiable Programming: Shift/Reset the Penultimate Backpropagator. *Proc. ACM Program. Lang.* 3, ICFP, Article 96 (July 2019), 31 pages. https://doi.org/10.1145/3341700

[24] 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).* Association for Computing Machinery, New York, NY, USA, Article 18, 15 pages. https://doi.org/10.1145/3190508.3190551