You Only Linearize Once

Tangents Transpose to Gradients

ALEXEY RADUL, Google Research, USA
ADAM PASZKE, Google Research, Poland
ROY FROSTIG, Google Research, USA
MATTHEW J. JOHNSON, Google Research, USA
DOUGAL MACLAURIN, Google Research, USA

Automatic differentiation (AD) is conventionally understood as a family of distinct algorithms, rooted in two "modes"—forward and reverse—which are typically presented (and implemented) separately. Can there be only one? Following up on the AD systems developed in the JAX and Dex projects, we formalize a decomposition of reverse-mode AD into (i) forward-mode AD followed by (ii) unzipping the linear and non-linear parts and then (iii) transposition of the linear part.

To that end, we define a (substructurally) linear type system that can prove a class of functions are (algebraically) linear. Our main results are that forward-mode AD produces such linear functions, and that we can unzip and transpose any such linear function, conserving cost, size, and linearity. Composing these three transformations recovers reverse-mode AD. This decomposition also sheds light on checkpointing, which emerges naturally from a free choice in unzipping `let` expressions. As a corollary, checkpointing techniques are applicable to general-purpose partial evaluation, not just AD.

We hope that our formalization will lead to a deeper understanding of automatic differentiation and that it will simplify implementations, by separating the concerns of differentiation proper from the concerns of gaining efficiency (namely, separating the derivative computation from the act of running it backward).

CCS Concepts: • **Mathematics of computing → Automatic differentiation**.

Additional Key Words and Phrases: automatic differentiation, decomposition, transpose, partial evaluation

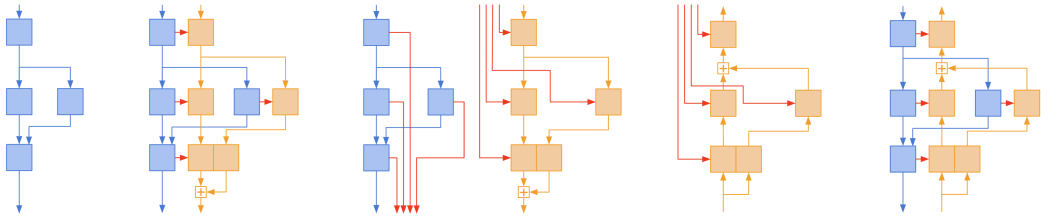
1 INTRODUCTION

Automatic differentiation (AD) powers not only deep machine learning, but also applications in broader numerical computing, from robotics to molecular dynamics to weather simulation. For example, in deep learning the task of choosing how to set the billions of parameters of a neural network is almost always attacked with AD: a programmer writes a prediction function representing how the network maps parameters and example inputs to predicted outputs, as well as a scalar-valued loss function penalizing deviations from expected outputs to predicted ones. Then to incrementally improve a given setting of the parameters, the programmer need only ask an AD system for the gradient of the loss with respect to the parameters on some example inputs and outputs, then update the parameters by adding a small negative multiple of that gradient. Repeat a million times, et voilà, la singularité est proche!

AD is traditionally organized into two *modes*, forward-mode and reverse-mode. Each mode has its advantages and applications. But most systems only implement one mode, or if both are implemented, the implementations are almost entirely separate.

Do they need to be? After all, reverse-mode AD is similar to forward-mode AD, except one computes the derivative part backward. We take this description literally: Our goal is to separate differentiation proper from direction reversal, and identify a clear boundary between them. The "direction reversal" is *transposition*, an interesting and potentially useful code transformation in its own right. The boundary between differentiation and transposition consists of provably linear

Authors' addresses: Alexey Radul, axch@google.com, Google Research, USA; Adam Paszke, apaszke@google.com, Google Research, Poland; Roy Frostig, frostig@google.com, Google Research, USA; Matthew J. Johnson, mattjj@google.com, Google Research, USA; Dougal Maclaurin, dougalm@google.com, Google Research, USA.



(a) f (b) forward derivative (c) forward derivative, unzipped (d) derivative, transposed (e) reverse derivative

Fig. 1. Reverse-mode automatic differentiation as forward differentiation, unzipping, and transposition.

functions: linearity (proven by our type system) is a sufficient condition for transposability, and we show that automatic derivatives are provably linear.

This simplifies implementations of reverse-mode AD (see Figure 1). The actual differentiation is the province of the widely known and relatively simple (and covariant!) forward-mode transformation. Reverse mode is derived therefrom by a separate transformation that now must only confront linearity, not derivatives. The notion of linearity we formalize here serves as a simpler and clearer (and mechanically verifiable) input invariant for transposition than one of the form "this program was generated by forward-mode AD, so it involves no unacceptable thing X."

This same architecture also makes AD systems easier for a user to extend. A user wishing to supply an efficient custom derivative for some function need only supply the forward (or reverse) derivative; as long as it is provably linear, the reverse (resp., forward) derivative can be derived automatically.

Reverse-mode automatic differentiation is already implemented in the proposed way in two systems we know of: JAX[1] [Bradbury et al. 2018; Frostig et al. 2018] and Dex [Paszke et al. 2021]. Our hope is to make the technique easier to maintain within these systems, by formalizing the provable linearity invariant; and to make it more widely accessible outside of them by describing it on a simple, stand-alone object language. We also hope to pave the way for exposing transposition as a user-accessible higher-order function in its own right, by clarifying its input and output invariants.

More concretely, our contributions are that:

- We explicitly decompose reverse-mode automatic differentiation into forward mode (Section 5), a new unzipping transformation (Section 6), and transposition (Section 7).
- We introduce a new linearly-typed intermediate language for linear computations (Section 4), whose type system captures the provable linearity invariant. We call the language "Linear A."
- We define and prove correctness for unzipping and transposition, and we prove that they preserve work, code size, and provable linearity. Together with the known results on correctness and work- and size-preservation for forward mode, this implies correctness and work- and size-preservation of reverse mode when implemented this way.[2]

2 PRELIMINARIES

2.1 Mathematical Differentiation Operations Corresponding to AD's Two Modes

At its core, AD brings two particular differentiation operations from mathematical functions to programming language functions. These two operations correspond to the forward and reverse

[1]The `jax.grad` function is exactly a composition of three functions that implement the steps we outline in this work.
[2]In our cost model, transposition and unzipping preserve work exactly, and program size up to a constant factor. Forward mode, as usual, only preserves work up to a constant factor.

modes. The first operation, the *Jacobian-vector product (JVP)*, underlies forward-mode and answers the question: if I perturb my input x a bit in some direction v, how does my output change? More precisely, given a sufficiently nice mathematical function $f : \mathbb{R}^n \to \mathbb{R}^m$, we can define the Jacobian at $x \in \mathbb{R}^n$ as the linear map $\partial f(x) : \mathbb{R}^n \to \mathbb{R}^m$ such that

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

We then say a *Jacobian-vector product for* f is the mapping

$$(x, v) \mapsto (f(x),\ \partial f(x)(v)).$$

This definition is compositional, in that the JVP of $f \circ g$ can be expressed by applying the JVP of f to the result of the JVP of g. It also allows for efficient evaluation, as we make precise in the sequel.

The second core mathematical operation represented in automatic differentiation is the *vector-Jacobian product (VJP)*, which underlies reverse-mode. It answers a more subtle question: given a linear function on small perturbations to a function's output, what is the corresponding linear function on small perturbations to the input? That is, for some fixed x, given a vector $u \in \mathbb{R}^m$, find the vector $w \in \mathbb{R}^n$ such that

$$\langle u,\ \partial f(x)(v)\rangle = \langle w,\ v\rangle, \qquad \forall v \in \mathbb{R}^n,$$

where $\langle \cdot, \cdot \rangle$ denotes the standard inner product. This mapping from u to w is in fact linear, so we can define a new linear map $\partial f(x)^T : \mathbb{R}^m \to \mathbb{R}^n$ by

$$\langle u,\ \partial f(x)(v)\rangle = \langle \partial f(x)^T(u),\ v\rangle.$$

We then say the *vector-Jacobian product for* f is the mapping

$$x \mapsto (f(x),\ u \mapsto \partial f(x)^T(u)).$$

This choice of definition is also compositional: the VJP of $f \circ g$ can be expressed by composing the VJPs of f and g, though the linear functions must be composed in reverse order as $\partial g(x)^T \circ \partial f(x)^T$. Hence the name "reverse-mode"!

The VJP is the more useful operation for gradient-based optimization, like in training neural networks. That's because with one VJP we compute the direction in parameter space orthogonal to the loss's local level sets. That is, for a function $f : \mathbb{R}^n \to \mathbb{R}$, the gradient is simply $\nabla f(x) = \partial f(x)^T(1) \in \mathbb{R}^n$. The gradient can also be computed using forward-mode, but then it requires n evaluations, losing the asymptotic efficiency of AD.

The implementation of reverse-mode is often considered much harder than that of forward-mode, with little shared code between the two. Yet, the "real work" of differentiation is embedded in the linear functions $\partial f(x)$ and $\partial f(x)^T$, which are just transposes of each other.

2.2 Linearity

Linearity is the key to relating forward and reverse modes. The word "linear" actually has two relevant meanings when discussing a programming language for numerical computations:

1. From substructural logic and linear type systems, a subprogram $P(x)$ is *substructurally linear* if it uses its argument x exactly once.
2. From linear algebra, a (mathematical) function $f(x)$ is *algebraically linear* if it is a vector space homomorphism, i.e., if $f(x + y) = f(x) + f(y)$ and $f(cx) = cf(x)$ for scalar c.

These two notions are not named the same by accident. Indeed, when is a polynomial, written as a sum of products of variables (no exponentiation), algebraically linear in x? Exactly when every nonzero monomial term uses x exactly once. This is how, when we get to Theorem 4.1, we will be able to prove algebraic linearity using an appropriately designed substructural type system.

The Jacobian-vector product (JVP) map at a point is, by mathematical definition, linear algebraically, but it also turns out (as we will also show in Section 5) that its instantiation in AD is linear substructurally. That substructural linearity will, in turn, allow us to mechanically transpose (Section 7) the program for computing f into a (also substructurally linear) program that computes $f^{\mathcal{T}}$ with the same performance. This then recovers reverse-mode AD, which computes the vector-Jacobian-product (VJP) map.

Part of the point of this exercise is that the transposition transformation we will define in Section 7 applies to all substructurally linear functions. Substructurally linear functions thus form an abstraction boundary between differentiation and transposition; and transposition (of user-defined functions) could then be exposed as a code transformation directly.

Note that there are algebraically linear functions that our type system will not accept as substructurally linear, and on which our transposition transformation would fail. For example, the function $f(x) = (x * x)/x$ is linear in x, but it cannot be typed as linear in Linear A as written. Moreover, attempting to transpose such a function as we do, by recurring on the subexpressions, is doomed to failure—the subexpression $\ldots/x$ is not linear in x in isolation, so has no transpose. Fortunately, as we prove in Section 5, automatic differentiation never produces such functions.

3 NOTATION REFERENCE

By convention, we use an **over-dot** to name linear terms: $\dot{v}$. The goal is to evoke the conventional physics notation for derivatives, since automatic differentiation is our main topic. Formally, however, we give the dot itself no structured meaning. So, the variable $\dot{v}$ is just a different variable from v; any relationship it may have to v emerges from our program transformations, rather than from how we write it.

The **double-dot** $\ddot{v}$ connotes co-tangents (not double derivatives), i.e., linear terms that appear in a transposed function. The nomenclature is from the reverse phase of reverse-mode AD.

The **underline** $\underline{v}$ means "zero or more of these elements"; we put it below rather than above to avoid clashing with dots: $\underline{\dot{v}}$. When the same thing appears in different underlined expressions in the same context, they are parallel. For instance, when a rule mentions both $\underline{v}$ and $\underline{v : \tau}$, those lists are the same length.

The **semicolon** $(x; y)$ separates non-linear entities (on the left) from linear ones (on the right).

The **angle brackets** $\langle x, y \rangle$ mean dot product when defining the meaning of transposition, and delimit transposition environments when explaining its implementation.

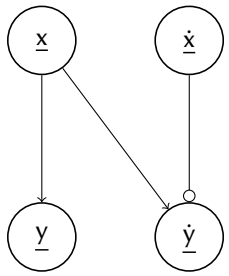
The **square brackets** $e[\underline{x}; \underline{y}]$ mean evaluating an expression with values $\underline{x}$ and $\underline{y}$ bound to its free non-linear and linear variables respectively.

4 LANGUAGE

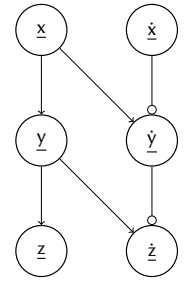
We introduce Linear A, a model language of *indexed linear* computations. The main idea in Linear A is that the syntax marks which values are supposed to be linear (substructurally and therefore algebraically) and which are not. Non-linear values can be computed arbitrarily (and may happen to be algebraically linear), but may not depend on linear values. Linear values, on the other hand, must be computed linearly from linear inputs, but may depend on non-linear values through scaling. This leads to the indexed-linear pattern of data flow shown in Figure 2.

For example, the function $g(x; y) = (x^2; x^2 y)$ is indexed linear. The first result of g depends non-linearly on x and does not depend on y at all; whereas the second result depends linearly on y, but the specific linear function of y is indexed by x. If we tag x and the first result as "non-linear", and y and the second result as "linear", g will type-check as a valid function of Linear A.[3]

[3]It will also type-check if we tag both inputs and both outputs "non-linear," but not with any other tagging.



(a) Expression e on variables $\underline{x}; \underline{\dot{x}}$ with results $\underline{y}; \underline{\dot{y}}$



(b) Sequential composition of expressions e_1, e_2

Fig. 2. Dataflow diagrams of (a) a general Linear A expression e, and (b) sequential composition of such expressions. The non-linear results of an expression (left) depend only only on its non-linear free variables, whereas the linear results (right, dotted) depend non-linearly on the non-linear free variables and linearly on the linear free variables. Composition preserves this (non-)dependence pattern.

The Linear A language is not designed as a user-facing language, but rather as a sublanguage that a given instance of automatic differentiation operates on. To wit, if one seeks to differentiate some F at some point x, the value of x determines the control flow choices in the implementation of F, and the program corresponding to the path actually taken (sometimes called a "trace") becomes straight-line. Differentiating such straight-line trace programs is the core task of AD, and we restrict Linear A to be total, functional, and first-order in order to focus our attention on it. While a more expressive object language would certainly be preferable, the technique of AD by explicit transposition has not been previously described even in this restricted setting; in Section 10, we will touch on some obstacles to formalizing transposition on a Turing-complete object language.

Many differentiable array languages are DSLs of the same expressive power as Linear A, embedded in a host language that provides Turing-complete metaprogramming facilities. Furthermore, reverse-mode automatic differentiation of control flow constructs (including higher-order functions) cannot be performed without forming a dynamic trace of the computation, and the languages conventionally used in those traces have expressive power similar to Linear A. For example, Pearlmutter and Siskind [2008a] and Wang et al. [2019] represent the trace as a sequence of applications. Similarly, Krawiec et al. [2022] propose an AD system capable of differentiating a rich higher-order functional language, but the differentiation involves tracing an expression in a language almost identical to the linear fragment of Linear A. Finally, the language used internally in the JAX project Bradbury et al. [2018], which implements our presented method, is also very close to Linear A. The main differences are that JAX has a significantly larger number of mathematical primitives, as well as some limited control-flow operators (conditionals, and loops with a statically known trip-count). None of these are fundamental to the presented method and for that reason we elide them for simplicity. JAX additionally supports array types with statically known shapes, which are isomorphic to product types in Linear A. On the other hand, Linear A is slightly richer than JAX's internal language, as it allows nested let-bindings to have an arbitrary expression on the right hand side, not only immediate primitive applications.

Many details of Linear A are unconventional, so we include point-by-point rationales. We focus on the syntax in Section 4.1, then present and discuss the type system in Section 4.2, state our cost model in Section 4.3, and formally define our notion of indexed linearity in Section 4.4, where we also prove that the type system enforces it. Finally, Section 4.5 defines the natural linearity erasure transformation on Linear A, which we will occasionally need later.

Types

$\tau, \sigma, \pi ::= \mathbb{R}$	real scalar
$\mid \otimes \underline{\tau}$	tuple type

Values (extralinguistic ground constants)

$x, y, \dot{x}, \dot{y} ::=$ number in $\mathbb{R}$	real scalar
$\mid \otimes \underline{x}$	tuple

m, n-ary Expressions

$e^{m,n} ::= (\ \underline{v}; \underline{\dot{v}}\ )$	Multi-value return; $\lvert\underline{v}\rvert = m$, $\lvert\underline{\dot{v}}\rvert = n$
$\mid \texttt{let(}\ \underline{v : \tau}; \underline{\dot{v} : \dot{\tau}}\ \texttt{)} = e^{o,p}\ \texttt{in}\ e^{m,n}$	multi-value let; $\lvert\underline{v}\rvert = o$, $\lvert\underline{\dot{v}}\rvert = p$
$\mid \texttt{let(}\ \otimes \underline{w} : \otimes\underline{\tau}; \ \texttt{)} = v\ \texttt{in}\ e^{m,n}$	unpacking let for non-linear tuples; $\lvert\underline{w}\rvert = \lvert\underline{\tau}\rvert$
$\mid \texttt{let(}\ ; \otimes\underline{\dot{w}} : \otimes\underline{\dot{\tau}}\ \texttt{)} = \dot{v}\ \texttt{in}\ e^{m,n}$	unpacking let for linear tuples; $\lvert\underline{\dot{w}}\rvert = \lvert\underline{\dot{\tau}}\rvert$
$\mid f^{o,p}_{m,n}(\underline{v}; \underline{\dot{v}})$	o, p-in, m, n-out function application; $\lvert\underline{v}\rvert = o$, $\lvert\underline{\dot{v}}\rvert = p$

Fixed-arity Expressions

$e^{1,0} ::= u, v, w, z$	non-linear variable
$\mid l$	literal; $l \in \mathbb{R}$
$\mid \otimes(\underline{v})$	tuple constructor
$\mid \sin(v) \mid \cos(v) \mid \exp(v) \mid v_1 + v_2 \mid \ldots$	primitives for non-linear fragment
$e^{0,1} ::= \dot{u}, \dot{v}, \dot{w}, \dot{z}$	linear variable
$\mid \hat{0}_{\dot{\tau}}$	linear zero of type $\dot{\tau}$
$\mid \otimes(\underline{\dot{v}})$	tuple constructor
$\mid \dot{v}_1 + \dot{v}_2$	linear addition
$\mid v * \dot{v}$	right-linear multiplication
$e^{0,2} ::= \mathrm{dup}(\dot{v})$	explicit fan-out for linear values
$e^{0,0} ::= \mathrm{drop}(e^{m,n})$	explicit drop

Definitions

$d ::= \texttt{def}\ f^{m,n}_{o,p}(\ \underline{v : \tau}; \underline{\dot{v} : \dot{\tau}}\ ) \to (\underline{\tau}; \underline{\dot{\tau}})\ = e^{o,p}$	m, n-in, o, p-out function; $\lvert\underline{v}\rvert = m$, $\lvert\underline{\dot{v}}\rvert = n$, $\lvert\underline{\tau}\rvert = o$, $\lvert\underline{\dot{\tau}}\rvert = p$

Programs

$P ::= \epsilon \mid d, P$	list of function definitions

Environments

$\Gamma ::= \epsilon \mid v : \tau, \Gamma$	unordered type environment
$\Sigma ::= \epsilon \mid v \to w, \Sigma$	variable association map (for $\mathcal{J}$ in Section 5)
$\Delta ::= \langle\rangle \mid \langle v : \tau, \Delta\rangle$	ordered type environment (for $\mathcal{T}$ in Section 7)

Contexts (for $\mathcal{U}$ in Section 6)

$E ::= \Box$	null context
$\mid \texttt{let(}\ \underline{v : \tau}; \ \texttt{)} = e^{n,0}\ \texttt{in}\ E$	non-linear let context, $\lvert\underline{v}\rvert = n$
$\mid \texttt{let(}\ \otimes \underline{w} : \otimes\underline{\tau}; \ \texttt{)} = v\ \texttt{in}\ E$	non-linear unpack context
$\mid E_1, E_2$	composition of contexts, by hole-substitution

Fig. 3. Syntax of Linear A. Expressions are syntactically indexed by how many non-linear and linear results they return. A function name enters scope after its definition—no recursive function calls. The underline $\underline{v}$ means "zero or more of these elements"; we put it below rather than above to avoid clashing with dots: $\underline{\dot{v}}$.

4.1 Syntax

The syntax of Linear A appears in Figure 3. We make Linear A a first-order language by only permitting function definitions at the top level. We make Linear A total by disallowing recursion: function names are bound only after their definition, as given by the program order. We also syntactically mark which values are linear vs not. In our notation, this distinction shows up as returns, binders, and function application forms taking two lists of arguments (or formal parameters): the non-linear arguments before the semicolon and the linear ones after.

Our function $g(x; y) = (x^2; x^2 y)$ from the previous section would be written:

$$
\begin{aligned}
&\texttt{def}\ g^{1,1}_{1,1}(\ x : \mathbb{R}; \dot{y} : \mathbb{R}\ ) \to (\mathbb{R}; \mathbb{R})\ = \\
&\quad \texttt{let(}\ a : \mathbb{R}; \ \texttt{)} = x * x\ \texttt{in} \\
&\quad \texttt{let(}\ ; \dot{b} : \mathbb{R}\ \texttt{)} = a * \dot{y}\ \texttt{in} \\
&\quad (a; \dot{b})
\end{aligned}
$$

Note that $*$ (as well as $+$) are available as primitives to both linear and non-linear computations in Linear A, just with different typing rules.

4.1.1 Tuples and Multiple Returns. We distinguish between product types (written $\otimes$), which are first-class in Linear A, and multiple-value returns (written $(\underline{v}; \underline{\dot{v}})$), which are the only construct permitted to mention both linear and non-linear values together. An expression returning multiple values cannot be bound to one variable, but must instead be bound componentwise. We enforce this in the syntax by indexing every expression $e^{m,n}$ by the number m of non-linear results it returns and the number n of linear results it returns. As a short-hand, we write e for a one-value non-linear expression, and $\dot{e}$ for a one-value linear expression. Products, in contrast, may be bound to variables, but must contain only linear or only non-linear values.

Multiple-value returns are a somewhat unusual syntactic choice. In our case, they are convenient, because our transposition transformation of Section 7 will be running expressions "backwards". Since an expression is free to reference more than one variable from its environment, its transpose must be able to return more than one result; and it's more elegant to do so directly in the syntax.

4.1.2 A-Normal Form. The Linear A syntax enforces administrative normal form [Sabry and Felleisen 1992] on expressions. Where needed, A-normal form can be enforced by standard techniques; we ask the reader to indulge us when we write some transformation results without explicitly introducing all the intermediate variables that our syntax technically requires. We do this for clarity and brevity. Under the same aegis, we also only allow primitives to be unary or binary, and all primitives except dup and drop return exactly one result. We likewise occasionally abuse notation and write $(; e, \underline{v})$ where e may return multiple results. The intended meaning is an expression that executes e and returns all of its results, and also the variables $\underline{v}$. This can be arranged within the syntax of Linear A by allocating fresh names to briefly hold the results of e.

A-normal form does make our work preservation results easier. To wit, it's possible and reasonable to implement a differentiate-unzip-transpose AD system that operates directly on compound expressions. Such a system would, however, need to be careful to introduce temporaries to avoid duplicating code expressions (and therefore work); for example in rule JPrimMul in Figure 7. A-normal form saves us from that because the intermediate names are already present.

4.1.3 Asymmetric Linear Multiply. Linear computations in Linear A model real multiplication as linear in the right-hand argument and non-linear in the left-hand argument. (This materializes in rule TypeLinMul in Figure 4 checking the left argument in the non-linear environment and the right argument in the linear environment.) Which is of course immaterial, because real multiplication commutes, but we intentionally pick one to keep the linearity of variables syntactically apparent. Of course, multiplication is actually *multi*-linear, but we leave explicitly modeling this as an open avenue for future research.

4.2 Typing

The type system for Linear A appears in Figure 4. It is a conventional substructural type system that enforces the indexed linearity that Linear A is trying to capture.

4.2.1 Substructural Typing. The type system is linear on the linear fragment of Linear A, enforcing that every linear variable is used exactly once, except where duplicated or dropped by explicit dup or drop operations. We track linear duplications and deletions because they transpose to addition and zero, respectively. Our earlier analogy between algebraic and substructural linearity of a polynomial appears here in the rules TypeLinPlus, which requires both arguments to be linear, and TypeLinMul, which requires exactly the right-hand argument to be linear.

$$\boxed{\Gamma; \dot{\Gamma} \vdash e^{n,m} : (\underline{\tau}; \underline{\dot{\tau}}),\ |\underline{\tau}| = n,\ |\underline{\dot{\tau}}| = m}$$

$$\frac{}{\underline{v : \tau}; \underline{\dot{v} : \dot{\tau}} \vdash (\underline{v}; \underline{\dot{v}}) : (\underline{\tau}; \underline{\dot{\tau}})}\ \textsc{TypeRet}$$

$$\frac{\Gamma_1; \dot{\Gamma}_1 \vdash e_1 : (\underline{\tau}; \underline{\dot{\tau}}) \quad \Gamma_2, \underline{v : \tau}; \dot{\Gamma}_2, \underline{\dot{v} : \dot{\tau}} \vdash e_2 : (\underline{\sigma}; \underline{\dot{\sigma}})}{\Gamma_1 \cup \Gamma_2; \dot{\Gamma}_1 \uplus \dot{\Gamma}_2 \vdash \texttt{let(}\ \underline{v : \tau}; \underline{\dot{v} : \dot{\tau}}\ \texttt{)} = e_1\ \texttt{in}\ e_2 : (\underline{\sigma}; \underline{\dot{\sigma}})}\ \textsc{TypeLet}$$

$$\frac{\Gamma, \underline{w : \tau}; \dot{\Gamma} \vdash e : (\underline{\sigma}; \underline{\dot{\sigma}})}{\Gamma \cup \{v : \otimes \underline{\tau}\}; \dot{\Gamma} \vdash \texttt{let(}\ \otimes \underline{w} : \otimes \underline{\tau};\ \texttt{)} = v\ \texttt{in}\ e : (\underline{\sigma}; \underline{\dot{\sigma}})}\ \textsc{TypeUnpack}$$

$$\frac{\Gamma; \dot{\Gamma}, \underline{\dot{w} : \dot{\tau}} \vdash e : (\underline{\sigma}; \underline{\dot{\sigma}})}{\Gamma; \dot{\Gamma}, \dot{v} : \otimes \underline{\dot{\tau}} \vdash \texttt{let(}\ ; \otimes \underline{\dot{w}} : \otimes \underline{\dot{\tau}}\ \texttt{)} = \dot{v}\ \texttt{in}\ e : (\underline{\sigma}; \underline{\dot{\sigma}})}\ \textsc{TypeLinUnpack}$$

$$\frac{\texttt{def}\ f\texttt{(}\ \underline{v : \tau}; \underline{\dot{v} : \dot{\tau}}\ \texttt{)} \to (\underline{\sigma}; \underline{\dot{\sigma}})\ \texttt{=}\ e}{\underline{v : \tau}; \underline{\dot{v} : \dot{\tau}} \vdash f(\underline{v}; \underline{\dot{v}}) : (\underline{\sigma}; \underline{\dot{\sigma}})}\ \textsc{TypeApp}$$

$$\frac{}{v : \tau; \varepsilon \vdash v : (\tau;)}\ \textsc{TypeVar} \qquad \frac{}{\varepsilon; \varepsilon \vdash l : (\mathbb{R};)}\ \textsc{TypeLit}$$

$$\frac{}{v : \mathbb{R}; \varepsilon \vdash \sin(v) : (\mathbb{R};)}\ \textsc{TypePrim1} \qquad \frac{}{\{v_1 : \mathbb{R}\} \cup \{v_2 : \mathbb{R}\}; \varepsilon \vdash v_1 + v_2 : (\mathbb{R};)}\ \textsc{TypePrim2}$$

$$\frac{}{\cup\{\underline{v : \tau}\}; \varepsilon \vdash \otimes \underline{v} : (\otimes \underline{\tau};)}\ \textsc{TypeTup} \qquad \frac{}{\varepsilon; \underline{\dot{v} : \dot{\tau}} \vdash \otimes \underline{\dot{v}} : (; \otimes \underline{\dot{\tau}})}\ \textsc{TypeLinTup}$$

$$\frac{}{\varepsilon; \dot{v} : \dot{\tau} \vdash \dot{v} : (; \dot{\tau})}\ \textsc{TypeLinVar} \qquad \frac{}{\varepsilon; \varepsilon \vdash \dot{0}_{\dot{\tau}} : (; \dot{\tau})}\ \textsc{TypeLinZero}$$

$$\frac{}{\varepsilon; \dot{v}_1 : \dot{\tau}, \dot{v}_2 : \dot{\tau} \vdash \dot{v}_1 + \dot{v}_2 : (; \dot{\tau})}\ \textsc{TypeLinPlus} \qquad \frac{}{v_1 : \mathbb{R}; \dot{v}_2 : \dot{\tau} \vdash v_1 * \dot{v}_2 : (; \dot{\tau})}\ \textsc{TypeLinMul}$$

$$\frac{}{\varepsilon; \dot{v} : \dot{\tau} \vdash \mathrm{dup}(\dot{v}) : (; \dot{\tau}, \dot{\tau})}\ \textsc{TypeDup} \qquad \frac{\Gamma; \dot{\Gamma} \vdash e : (\underline{\tau}; \underline{\dot{\tau}})}{\Gamma; \dot{\Gamma} \vdash \mathrm{drop}(e) : (;)}\ \textsc{TypeDrop}$$

$$\frac{\boxed{\vdash f} \qquad \underline{v : \tau}; \underline{\dot{v} : \dot{\tau}} \vdash e : (\underline{\sigma}; \underline{\dot{\sigma}})}{\vdash \texttt{def}\ f\texttt{(}\ \underline{v : \tau}; \underline{\dot{v} : \dot{\tau}}\ \texttt{)} \to (\underline{\sigma}; \underline{\dot{\sigma}})\ \texttt{=}\ e}\ \textsc{TypeDef}$$

Fig. 4. Linear A typing rules. This is a standard simply-typed system, except that it enforces that all non-linear variables are used at least once, and all linear variables are used exactly once, up to the explicit drop and dup operations. The TypePrim rules can be extended to a larger set of primitive differentiable maps.

Less conventionally, we also enforce that every non-linear variable is used at least once (by an explicit drop if needed). Thus even the non-linear fragment of Linear A requires a substructural (if not *linear*, per se) type system. We track drop of non-linear variables because we have to charge for it in our cost model (see Section 4.3.1 for a more complete rationale); we do not track non-linear duplication because we do not charge for dup.

Since our requirement for non-linear variables is to use each at least once, we often pattern-match a non-linear type environment as $\Gamma_1 \cup \Gamma_2$ (e.g., in TypeLet) or $\cup \Gamma_i$, implying "Γ_i contains exactly the free variables of e_i; the incoming environment Γ must be the (not necessarily disjoint) union of the Γ_i." Similarly, when speaking of the linear environment, we write $\dot{\Gamma}_1 \uplus \dot{\Gamma}_2$ to imply that in this case the union must be disjoint. In the end, the actual use of variables is enforced by each leaf rule requiring that the incoming environment contain exactly the variables referenced by that leaf and no others. In practice, one would implement such a type system by inspecting the free variables of subexpressions rather than by guessing and checking a partition of the environment, but we elide that consideration from the rules for the sake of brevity.

4.2.2 Polymorphism. We define linear zero, linear addition, and linear multiplication to be type-indexed—the same operation will type-check with different types in different places as needed (rules TypeLinZero, TypeLinPlus, TypeLinMul). This is justified because every type that a linear variable can have defines a unique vector space isomorphic to $\mathbb{R}^n$ for some n. Operationally, the zero (respectively, summation and scaling) just happens elementwise, recurring into tuples as needed.

The reason to make linear operations type-indexed like this is that drop and dup are naturally type-indexed (i.e., drop can drop a value of any type), but they transpose to zero and addition, respectively. We could also type-index non-linear operations, but we don't need to, and it would cease to be justified if Linear A were extended to include sum types. Thus, TypeLit, TypePrim1 and TypePrim2 call for the $\mathbb{R}$ type.

4.3 Cost Model

The vast automatic differentiation literature often makes efficiency preservation claims, but fails to make the cost model precise. As it turns out, the cost model required for AD to preserve efficiency is not necessarily obvious. Here, we state *a* model explicitly, albeit one that has been specifically tailored to make our cost preservation claims hold. While somewhat unsatisfactory (e.g., we don't base it on a well-established abstract machine), we think it is still a worthwhile contribution, as it aids the reader in judging the way our method modifies the run-time more precisely. And, apart from the two finer points we elaborate on below, it could be considered a fairly standard model for an eager language with call-by-value semantics.

Specifically, we choose the following call-by-value cost model for Linear A:

- Every non-linear primitive costs 1.
- Linear addition and linear multiplication each cost 1 per scalar present in the result.
- The drop operation costs the cost of its subexpression, plus 1 for every scalar of real type (linear or non-linear) present in the argument.
- Applying a function costs evaluating its body on the values of its arguments.
- The `let` form costs evaluating its bound expression and then evaluating its body. This models call-by-value evaluation.
- All other syntactic forms cost 0.

Two specific choices bear some elaboration:

4.3.1 Costly Dropping of Variables. Why do we charge for drop, when programmers generally think that doing nothing with a variable is free? Because we need it for work to be preserved.

The fundamental problem with work preservation of reverse-mode AD is that fan-out (i.e., dup) is intuitively costless, but transposes to addition, which is intuitively costly. We could solve that problem by charging for dup of linear variables, but then forward differentiation wouldn't be (locally) work-preserving unless we charged for dup of non-linear variables as well. Instead, we follow the observation that charging for dup should have no asymptotic effect, because usually both duplicates would participate in some other costly operation. In order to make it so, we must charge for drop (of both linear and non-linear variables), to make it into a costly operation; and doing that indeed suffices for the work preservation proofs to go through. Another reason why charging for drop instead of dup is more natural, is that drop should be uncommon, or even absent from non-pathological programs since it is only necessary in presence of dead code.

4.3.2 Costless Tuples. Why do we not charge for constructing or unpacking tuples, or binding or referencing variables, which would correspond to costly allocation and access of memory in a real implementation? The main reason is that reverse-mode AD actually does increase the asymptotic memory footprint of a program. In our system, this is visible in the ULet and UDef rules in

Section 6, which drastically increase the lifetimes of intermediate non-linear variables (including across function calls).

4.4 Linearity

Every well-typed Linear A expression is *indexed linear* when viewed as a function from its environment to its return value(s). We illustrated the dependence and linearity structure in Figure 2; now we formalize and prove it in Theorem 4.1. Specifically, a Linear A expression e defines a collection of algebraically linear functions—from its free linear variables to its linear outputs—indexed by values of its free non-linear variables. The indexing is non-trivial in general because linear multiplication takes a non-linear argument on the left.

Theorem 4.1. ***Linear A expressions are indexed linear.*** *Consider an expression e of Linear A. Consider also any set of well-typed values $\underline{x}$ for the non-linear variables $\underline{v}$ free in e, and any two sets of well-typed values $\underline{\dot{x}}$ and $\underline{\dot{y}}$ for the linear variables $\underline{\dot{v}}$ free in e; and any scalar c. We write $e[\underline{x}; \underline{\dot{x}}]$ for the result of evaluating e with $\underline{v}$ bound to $\underline{x}$ and $\underline{\dot{v}}$ bound to $\underline{\dot{x}}$. We use subscripts as indexing here, so $e[\underline{x}; \underline{\dot{x}}]_k$ is the k-th result of evaluating e on $\underline{x}$ and $\underline{\dot{x}}$. Then:*

1. *The total work of evaluating $e[\underline{x}; \underline{\dot{x}}]$ is independent of the values $\underline{x}$ and $\underline{\dot{x}}$;*
2. *The non-linear results are independent of $\underline{\dot{x}}$, i.e., $e[\underline{x}; \underline{\dot{x}}]_k = e[\underline{x}; \underline{\dot{y}}]_k$ for each non-linear result k;*
3. *The linear results are linear in $\underline{\dot{x}}$, i.e., $e[\underline{x}; \underline{\dot{x}} + \underline{\dot{y}}]_l = e[\underline{x}; \underline{\dot{x}}]_l + e[\underline{x}; \underline{\dot{y}}]_l$ and $e[\underline{x}; \underline{c \cdot \dot{x}}]_l = c \cdot e[\underline{x}; \underline{\dot{x}}]_l$ for each linear result l.*

The equality, addition and scaling here are mathematical operations over reals, and we assume the denotation of arithmetic in Linear A to be carried out on infinite-precision reals. Given approximate (e.g., floating-point) semantics, the equalities might not be exact.

Note that Claim 1 of this theorem, while true for Linear A, is not future-proof. A variant of the language that supported (non-linear) conditionals or recursion would need to weaken Claim 1 to allow work (including termination) to depend on the non-linear values $\underline{x}$, though work should remain independent of the linear values $\underline{\dot{x}}$.

Proof. By structural induction, first on the program and then on the syntax of expressions and function bodies. Work-independence (Claim 1) follows immediately from the lack of conditional control flow in the language.

The proofs of Claims 2 and 3 are similar enough that we discuss them jointly. The interesting case is multi-value `let`, so we consider it first. The inductive hypothesis asserts that the non-linear values returned by the bound expression are independent of the linear free variables thereof, and the returned linear values satisfy the additive and scaling laws. Since the set of free variables of the body expression is a union of free variables of the entire expression and the let bound values, we can again invoke the inductive hypothesis to arrive at the desired conclusion.

Tuple unpacking is entirely analogous to the multi-value `let`. Function application follows directly from induction on the function body.[4] The drop form is trivial, and all other forms can be verified directly because they do not have subexpressions. In particular, indexed linearity depends upon the only primitive operations available to the linear fragment of Linear A being zero, plus, and scaling by a non-linear constant; and the presence of actually non-linear primitives in the non-linear fragment of Linear A is what prevents us from strengthening the linearity claims to the non-linear variables. □

Corollary 4.2. *Every closed term in Linear A returns 0 for all its linear results.*

[4] This is why we need the outer induction on the whole program rather than just structural induction on the syntax of a single expression.

Proof. If the term e is closed, we can vacuously scale all of the free linear variables of e without changing the evaluation of e. All the linear results must therefore be 0. Formally, Claim 3 gives us $e[;]_l = 2e[;]_l = 0$ for every linear result position l of e. □

In particular, for the statement of Theorem 4.1 not to be vacuous, one must allow $\dot{\underline{x}}$ and $\dot{\underline{y}}$ to be non-zero without asking where those non-zero values came from. This is ok, though, because we are generally interested in *open* terms of Linear A, such as the bodies of Linear A functions. In that case, we can assume the non-zero linear values are provided externally through the substitution of free variables.

Lemma 4.3. *The work a Linear A function or expression does is at least the number of real scalars in its linear inputs, less the number of real scalars in its linear outputs.*

Proof. The only linear operations in Linear A that return fewer scalar results than they consume are + and drop, each of which costs 1 per net scalar consumed. Since all linear variables must be either consumed or returned, the result follows. □

We state this fact here because it's a curious general property of our cost model. We will use it only to argue that eliminating useless code (e.g., by short-circuiting in the rule for transposing $\mathrm{drop}(e)$) does not increase work (!).

4.5 Linearity Erasure

Because we will occasionally need it, we define a linearity erasure transform $\mathcal{L}$ on Linear A. Linearity erasure is just what it sounds like: remove the linearity information by turning all linear variables and expressions into non-linear ones. Syntactically, this just means

- For every type at which linear addition, multiplication, and zero are used in the program, define a fresh function that carries that operation out in the non-linear fragment on Linear A (using unpacking and repacking as needed);
- Replace every linear addition, multiplication, and zero with a call to the corresponding function;
- Rename every reference to a variable produced by dup to refer to the input of that dup instead;
- Remove all dup operations; and
- Move all the semicolons all the way to the right.

In the interest of space, we do not give rules for this transform.

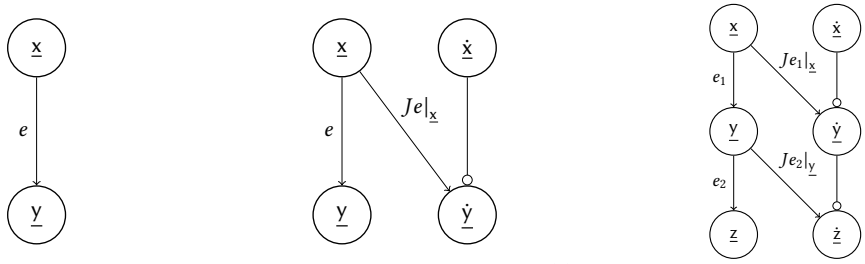
Lemma 4.4. *Linearity erasure preserves the semantics of an expression, and preserves work in our cost model.*

Proof. By inspection. □

5 AUTOMATIC DIFFERENTIATION

Our first step is a code transformation $\mathcal{J}$ performing forward-mode automatic differentiation[5]. Specifically, $\mathcal{J}$ transforms the non-linear fragment of Linear A into the full language as follows. For any purely non-linear e in Linear A, and input $\underline{x}$, $\mathcal{J}(e)[\underline{x};\dot{\underline{x}}]$ has non-linear results computing the same output as $e[\underline{x};]$, and linear results computing the directional derivative of e at the point $\underline{x}$ in the direction $\dot{\underline{x}}$. We illustrate the data flow produced by $\mathcal{J}$ in Figure 5. Here we view the expression e as a function from its free variables $\underline{v}$ to its results. A code example appears in Figure 6.

[5] $\mathcal{J}$ corresponds to the `jax.jvp` transform from JAX.



(a) Original expression e (b) Differentiated expression $\mathcal{J}(e)$ (c) Sequence of $\mathcal{J}(e)$s

Fig. 5. Dataflow diagrams of (a) a purely non-linear expression in Linear A and (b) its derivative, plus (c) a derivative made of a sequential composition stacked vertically. The primal (non-linear) results of a derivative depend only only on its primal (non-linear) inputs, whereas the tangent results depend non-linearly on the primal (non-linear) inputs and linearly on the tangent (linear) inputs. Note the distinction between $\mathcal{J}$, the code transform we define, and J, the Jacobian of an expression e.

```
def f^{1,0}_{1,0}( u : ℝ; ) → (ℝ;) =
    let( v : ℝ; ) = sin(u) in
    let( w : ℝ; ) = − v in
    (w;)
```

$\xrightarrow{\mathcal{J}}$

```
def f^{J 1,1}_{1,1}( u : ℝ; u̇ : ℝ ) → (ℝ;ℝ) =
    let( v : ℝ; ) = sin(u) in
    let( dv : ℝ; ) = cos(u) in
    let( ; v̇ : ℝ ) = dv ∗ u̇ in
    let( w : ℝ; ) = − v in
    let( ; ẇ : ℝ ) = − v̇ in
    (w; ẇ)
```

Fig. 6. Example of forward differentiation with $\mathcal{J}$, transforming $f(u) = -\sin(u)$ on the left into $\mathcal{J}(f)$, which computes both the original f and its directional derivative. The input function f must be coded in Linear A as all non-linear. The result uses both the linear and non-linear fragments of Linear A.

To compute a forward derivative end-to-end, we need to supply the initial direction of differentiation (1 for an $\mathbb{R} \to \mathbb{R}^m$ primal function). Since we cannot write a non-zero linear literal in well-typed Linear A program, we have to use the linearity erasure transform $\mathcal{L}$ from Section 4.5. For $F : \mathbb{R} \to \mathbb{R}^m$,

$$F(x), \frac{d}{dx}F(x) = \mathcal{L}(\mathcal{J}(F))(x, 1;).$$

Part of the point of this whole exercise is that $\mathcal{J}$ will be very familiar to students of automatic differentiation; it's just a forward-mode transformation that computes tangents in tandem with primals. The rules in Figure 7 are completely standard, with the one wrinkle that the derivative is emitted into the linear fragment of Linear A. Ergo, the result being well-typed (substructurally) in Linear A means that we have a proof that the derivative of a function is algebraically linear (with respect to the direction in which the derivative is taken).

This is the first step in our separation of concerns: differentiation proper is confined to the relatively simple $\mathcal{J}$, whereas arranging to run the derivative backward to obtain reverse-mode AD is the province of $\mathcal{U}$ and $\mathcal{T}$ (in Section 6 and Section 7, respectively). The latter two know only about linearity, not about differentiation, nor about any special relationship between the non-linear (primal) and the linear (tangent) computations. The type system of Linear A is the abstraction boundary that lets differentiation and transposition be implemented independently.

$$\boxed{\mathcal{J}(\Sigma \mid e) \rightsquigarrow e'}$$

$$\frac{\underline{v} \text{ distinct}}{\mathcal{J}(\underline{v} \to \dot{v} \mid (\underline{v};)) \rightsquigarrow (\underline{v}; \underline{\dot{v}})}\ \textsc{JRet}$$

$$\frac{\begin{array}{c}\mathcal{J}(\Sigma_1, \underline{u \to \dot{w}} \mid e_1) \rightsquigarrow e_1' \quad \mathcal{J}(\Sigma_2, \underline{u \to \dot{z}}, \underline{v \to \dot{v}} \mid e_2) \rightsquigarrow e_2' \\ \dot{v}, \underline{\dot{w}}, \underline{\dot{z}} \text{ fresh} \quad \underline{u : \tau} \text{ free in both } e_1 \text{ and } e_2\end{array}}{\begin{array}{l}\mathcal{J}(\Sigma_1 \uplus \Sigma_2 \uplus \{\underline{u \to \dot{u}}\} \mid \texttt{let(}\ \underline{v : \sigma};\ \texttt{)} = e_1 \texttt{ in } e_2) \\ \rightsquigarrow \texttt{let(}\ ; \underline{\dot{w} : \tau}, \underline{\dot{z} : \tau}\ \texttt{)} = (; \mathrm{dup}(\dot{u}))\ \texttt{in let(}\ \underline{v : \sigma}; \underline{\dot{v} : \sigma}\ \texttt{)} = e_1' \texttt{ in } e_2'\end{array}}\ \textsc{JLet}$$

$$\frac{\mathcal{J}(\Sigma, \underline{w \to \dot{w}} \mid e) \rightsquigarrow e' \quad \underline{\dot{w}} \text{ fresh} \quad v \text{ not free in } e}{\begin{array}{l}\mathcal{J}(\Sigma, v \to \dot{v} \mid \texttt{let(}\ \otimes \underline{w} : \otimes \underline{\tau}\ \texttt{)} = v \texttt{ in } e) \\ \rightsquigarrow \texttt{let(}\ \otimes \underline{w} : \otimes \underline{\tau}\ \texttt{)} = v \texttt{ in let(}\ \otimes \underline{\dot{w}} : \otimes \underline{\tau}\ \texttt{)} = \dot{v} \texttt{ in } e'\end{array}}$$

$$\frac{\underline{v} \text{ distinct}}{\mathcal{J}(\underline{v \to \dot{v}} \mid f(\underline{v};)) \rightsquigarrow f^{\mathcal{J}}(\underline{v}; \underline{\dot{v}})}\ \textsc{JApp} \qquad \frac{}{\mathcal{J}(v \to \dot{v} \mid v) \rightsquigarrow (v; \dot{v})}\ \textsc{JVar}$$

$$\frac{\underline{v} \text{ distinct}}{\mathcal{J}(\underline{v \to \dot{v}} \mid \otimes \underline{v}) \rightsquigarrow (\otimes \underline{v}; \otimes \underline{\dot{v}})}\ \textsc{JTup} \qquad \frac{}{\mathcal{J}(\varepsilon \mid l) \rightsquigarrow (l; 0_{\mathbb{R}})}\ \textsc{JLit}$$

$$\frac{}{\mathcal{J}(v \to \dot{v} \mid \sin(v)) \rightsquigarrow (\sin(v); \cos(v) * \dot{v})}\ \textsc{JPrimSin}$$

$$\frac{v_1, v_2 \text{ distinct}}{\mathcal{J}(v_1 \to \dot{v}_1, v_2 \to \dot{v}_2 \mid v_1 * v_2) \rightsquigarrow (v_1 * v_2; v_1 * \dot{v}_2 + v_2 * \dot{v}_1)}\ \textsc{JPrimMul}$$

$$\frac{\mathcal{J}(\Sigma \mid e) \rightsquigarrow e'}{\mathcal{J}(\Sigma \mid \mathrm{drop}(e)) \rightsquigarrow \mathrm{drop}(e')}\ \textsc{JDrop}$$

$$\boxed{\mathcal{J}(f) \rightsquigarrow f^{\mathcal{J}}}$$

$$\frac{\mathcal{J}(\underline{v \to \dot{v}} \mid e) \rightsquigarrow e' \quad \underline{\dot{v}} \text{ fresh}}{\mathcal{J}(\texttt{def } f\texttt{(}\ \underline{v : \tau};\ \texttt{)} \to (\underline{\sigma};\texttt{)} = e) \rightsquigarrow \texttt{def } f^{\mathcal{J}}\texttt{(}\ \underline{v : \tau}; \underline{\dot{v} : \tau}\ \texttt{)} \to (\underline{\sigma}; \underline{\sigma})\ \texttt{=}\ e'}\ \textsc{JDef}$$

Fig. 7. Forward-mode automatic differentiation in Linear A. The constructed expression e' emits primal and tangent results together as a multi-value return. The argument Σ maps primal variable names to the corresponding tangent variable names. Note in the result of rule JPrimMul, the multiplication on the left is non-linear, whereas the two multiplications on the right are right-linear. Several near-redundant rules are omitted; see text.

5.1 Details

5.1.1 Non-linear Input. We define $\mathcal{J}$ to operate only on the purely non-linear subset of Linear A. This avoids perturbation confusion problems: all linear variables in Linear A refer to the same perturbation. We can still get nested AD by alternating $\mathcal{J}$ and the linearity erasure transform $\mathcal{L}$.

5.1.2 Introducing dup. The non-linear fragment of Linear A allows variables to be referenced multiple times, but the linear fragment does not. Ergo, whenever a non-linear variable is re-used, we must introduce dup operations on the corresponding linear variables. The JLet rule in Figure 7 spells this out: the rule detects which variables $\underline{u}$ are repeated; constructs the needed fresh variables $\underline{\dot{w}}, \underline{\dot{z}}$; introduces the duplication operations $\underline{\dot{w}, \dot{z} = \mathrm{dup}(\dot{u})}$; and takes care that the tangent of $\underline{u}$ in e_1 is $\underline{\dot{w}}$, whereas in e_2 it's $\underline{\dot{z}}$.

One instance seeming sufficient, Figure 7 omits similar manipulations for the other rules. The full system has a rule JUnpackB for the case where the tuple variable v does appear free in the body e; it differs from JUnpackA only in that the binding for v is propagated to the recursive call, and the tangent $\dot{v}$ is duplicated for its appearance in the transformed body. Likewise, JRet, JApp, JTup, and JPrimMul have variants that add the necessary dup calls if any of the non-linear $\underline{v}$ repeat.

5.1.3 Other Omissions. We also omit rules for the other non-linear primitives. They differ from JPrimSin and JPrimMul only in the derivatives they must insert into the result; we trust that a unary and a binary example are adequately clear. Finally, there are no rules for linear syntax because $\mathcal{J}$ operates on the non-linear fragment of Linear A.

5.1.4 Extra Non-linear Computations. Note that $\mathcal{J}$ in general introduces non-linear computations that are not needed for the original non-linear program—these are the actual partial derivatives. For example, in JPrimSin we see a new $\cos(v)$. This is bound to a non-linear variable, but used only by the linear result of $\mathcal{J}(\sin)$. Figure 1 elided these extra non-linear computations, hiding them in the red arrows between the blue primal blocks and orange derivative blocks. Our downstream unzipping transformation $\mathcal{U}$ treats them as part of the blue blocks, since they are non-linear.

5.1.5 Tangent Types. The rules in Figure 7 depend on assuming that any Linear A type τ is also a suitable type for its tangent. As we have defined Linear A to have only real scalars and nested tuples thereof, this proposition is costless; but the correct choice of tangent type becomes more interesting in a language with sum types. One conventional choice is to still use τ itself as the type of tangents to τ, though that loses the information that both the primal and the tangent value are necessarily the same variant. Retaining that information requires a dependent type system; we look forward to the community working out a satisfying one for this purpose.

5.2 Formalization

Our theorem about forward differentiation is conventional. The interesting thing about it is that the $\mathcal{J}$ transformation puts the derivative into the linear fragment of Linear A. We therefore have a proof obligation that automated derivatives are substructurally linear; but otherwise this is a repetition of known results that forward differentiation is correct, and work-preserving up to a constant factor.

Theorem 5.1. ***Automatic differentiation is total, correct, and work-preserving.*** *Consider an expression e, well-typed in $\Gamma;\{\}$ in the non-linear fragment of Linear A. Let $\underline{x}$ be well-typed values for the (necessarily non-linear) free variables $\underline{v:\tau}$ of e. Allocate fresh names $\underline{\dot{v}}$ of types $\underline{\tau}$ for the tangents, and let $\underline{\dot{x}}$ be well-typed values for them. Then:*

1. *$e' = \mathcal{J}(\underline{v \to \dot{v}} | e)$ exists and is well-typed in $\Gamma; \underline{\dot{v}:\tau}$ in Linear A;*
2. *e' does no more than a constant times more work than e. The constant does not depend on e, but does depend on the set of primitives and their derivative rules;*
3. *The non-linear outputs of $e'[\underline{x};\cdot]$ are equal to the outputs of $e[\underline{x};]$; and*
4. *The linear outputs of $e'[\underline{x};\underline{\dot{x}}]$ are equal to the matrix-vector product of the Jacobian of e at $\underline{x}$ with the vector $\underline{\dot{x}}$, provided this is true of the primitive derivatives.*

To define the Jacobian for Claim 4, we view the expression e as a function from its free variables $\underline{v}$ to its results.

We assume these claims hold for all the primitives in a given implementation of Linear A. In particular, the constant in Claim 2 may not actually be well-determined in a large and user-extensible AD system, because it depends on the cost behavior of every primitive derivative on every set of arguments. We content ourselves in this paper with proving that $\mathcal{J}$ composes those primitive

derivatives correctly, in the sense of introducing no correctness or work errors that are not present in the primitive set already.

Proof. The proof proceeds by structural induction on the program. We work out the JLet rule as a template for the others. The first subtlety is that there may be some non-linear variables, $\underline{u}$ in the notation, that are free in both the bound expression e_1 and the body e_2. To construct the parallel linear expression, we must insert explicit dups for their corresponding linear variables $\underline{\dot{u}}$. This is notated as `let( ; ẇ, ż ) = (; dup(u̇)) in` in JLet, implying one `let` and one dup per duplicated variable among the $\underline{\dot{u}}$. This duplication is also why the map from primal variables to their corresponding tangent variables may be non-trivial and needs to be maintained explicitly. Work preservation (Claim 2) follows because the dups introduced by JLet cost 0 in our cost model.

Non-linear correctness (Claim 3) follows from the inductive hypothesis because we reconstruct the non-linear part of the computation exactly in the result $\mathcal{J}(e)$. Linear correctness (Claim 4) is a calculation:

$$\begin{aligned}
\mathcal{J}(e)[\underline{x};\underline{\dot{x}}] &= \mathcal{J}(e_2)[\underline{x}_2, e_1[\underline{x};]; \underline{\dot{x}}_2, \mathcal{J}(e_1)[\underline{x}_1;\underline{\dot{x}}_1]] && \text{definition of } \mathcal{J}(e)\\
&= \left(Je_2|_{\underline{x}_2, e_1[\underline{x};]}\right)(\underline{\dot{x}}_2, \mathcal{J}(e_1)[\underline{x}_1;\underline{\dot{x}}_1]) && \text{induction on } e_2\\
&= \left(Je_2|_{\underline{x}_2, e_1[\underline{x};]}\right)\left(\underline{\dot{x}}_2, \left(Je_1|_{\underline{x}_1}\right)(\underline{\dot{x}}_1)\right) && \text{induction on } e_1\\
&= \left(Je|_{\underline{x}}\right)(\underline{\dot{x}}). && \text{chain rule on } e_2 \circ e_1
\end{aligned}$$

The chain rule is for the composition of multivariate functions $e_2 \circ e_1$ augmented with side inputs $\underline{x}_2$ and their tangents $\underline{\dot{x}}_2$. Here we somewhat abuse our own notation, writing $(Je|.)$ for the Jacobian of e at a point, $\mathcal{J}(e)[\cdot;\cdot]$ for the *linear* results of evaluating $\mathcal{J}(e)$, and $e[\cdot;]$ with no linear inputs for the *non-linear* results of evaluating e. The reader must also understand the index 1 to refer to variables free in e_1 and 2 to refer to those free in e_2, with any overlap among linear ones taken care of by the above-mentioned chain of `let` and dup.

Induction on the program (including previously processed definitions) comes in when checking JApp, because we rely on correctness of the transformed callee for correctness of the transformed call site. The handling of JUnpack is analogous to JLet, and the other rules are straightforward. □

6 UNZIPPING

We cannot directly transpose arbitrary Linear A programs. Why not? In the transposed result we are trying to run the non-linear part forward and the linear part backward. However, the first sub-expression of a general Linear A program may produce both linear and non-linear results. Do we run that subexpression before or after the remainder?

Fortunately, the non-linear part of a Linear A expression cannot depend on any of its linear inputs. Ergo, it should be possible to hoist it above the linear part (this section) to just transpose the latter (Section 7).

We begin by defining what we want:

Definition 6.1 (Linear B). Linear B is a sublanguage of Linear A with two restrictions:

- All functions and expressions of Linear B must return either all-linear or all-nonlinear results—no mixing.
- Expressions of Linear B that produce non-linear results must not read linear variables, not even to drop them.

$$\boxed{\mathcal{U}(e) \rightsquigarrow E; e'; \dot{e}'}$$

$$\frac{}{\mathcal{U}((\underline{v}; \underline{\dot{v}})) \rightsquigarrow \Box; (\underline{v};); (;\underline{\dot{v}})}\ \textsc{URet}$$

$$\frac{\mathcal{U}(e_1) \rightsquigarrow E_1; e_1'; \dot{e}_1' \quad \mathcal{U}(e_2) \rightsquigarrow E_2; e_2'; \dot{e}_2'}{\mathcal{U}(\texttt{let(}\ \underline{v : \tau}; \underline{\dot{v} : \dot{\tau}}\ \texttt{)} = e_1\ \texttt{in}\ e_2) \rightsquigarrow E_1, \texttt{let(}\ \underline{v : \tau};\ \texttt{)} = e_1'\ \texttt{in}\ E_2;\ e_2';\ \texttt{let(}\ ;\underline{\dot{v} : \dot{\tau}}\ \texttt{)} = \dot{e}_1'\ \texttt{in}\ \dot{e}_2'}\ \textsc{ULet}$$

$$\frac{\mathcal{U}(e) \rightsquigarrow E; e'; \dot{e}'}{\mathcal{U}(\texttt{let(}\ \otimes \underline{w} : \otimes\underline{\tau};\ \texttt{)} = v\ \texttt{in}\ e) \rightsquigarrow \texttt{let(}\ \otimes \underline{w} : \otimes\underline{\tau}\ \texttt{)} = v\ \texttt{in}\ E;\ e';\ \dot{e}'}\ \textsc{UUnpack}$$

$$\frac{\mathcal{U}(e) \rightsquigarrow E; e'; \dot{e}'}{\mathcal{U}(\texttt{let(}\ ;\otimes\underline{\dot{w}} : \otimes\underline{\dot{\tau}}\ \texttt{)} = \dot{v}\ \texttt{in}\ e) \rightsquigarrow E;\ e';\ \texttt{let(}\ \otimes \underline{\dot{w}} : \otimes\underline{\dot{\tau}}\ \texttt{)} = \dot{v}\ \texttt{in}\ \dot{e}'}\ \textsc{ULinUnpack}$$

$$\frac{\vdash f\texttt{.nonlin(}\cdot;\texttt{)} \to (\underline{\tau}, \sigma;) \quad \underline{u}, w \text{ fresh}}{\mathcal{U}(f(\underline{v}; \underline{\dot{v}})) \rightsquigarrow \texttt{let(}\ \underline{u : \tau}, w : \sigma;\ \texttt{)} = f\texttt{.nonlin(}\underline{v};\texttt{)}\ \texttt{in}\ \Box;\ (\underline{u};);\ f\texttt{.lin(}w; \underline{\dot{v}})}\ \textsc{UApp}$$

$$\frac{e \text{ non-linear with no sub-expressions}}{\mathcal{U}(e) \rightsquigarrow \Box; e; (;)}\ \textsc{ULeaf}$$

$$\frac{\dot{e} \text{ linear with no sub-expressions}}{\mathcal{U}(\dot{e}) \rightsquigarrow \Box; (;); \dot{e}}\ \textsc{ULinLeaf}$$

$$\frac{\mathcal{U}(e) \rightsquigarrow E; e'; \dot{e}'}{\mathcal{U}(\text{drop}(e)) \rightsquigarrow E; \text{drop}(e'); \text{drop}(\dot{e}')}\ \textsc{UDrop}$$

$$\boxed{\mathcal{U}(f) \rightsquigarrow f\texttt{.nonlin}, f\texttt{.lin}}$$

$$\frac{\mathcal{U}(e) \rightsquigarrow E; e'; \dot{e}' \quad \underline{w : \pi} \text{ are all the names bound by } \underline{v} \text{ and } E \text{ and read by } \dot{e}' \quad u \text{ fresh}}{\begin{array}{c}\mathcal{U}(\texttt{def}\ f\texttt{(}\ \underline{v : \tau}; \underline{\dot{v} : \dot{\tau}}\ \texttt{)} \to (\underline{\sigma}; \underline{\dot{\sigma}})\ \texttt{=}\ e) \rightsquigarrow \texttt{def}\ f\texttt{.nonlin(}\ \underline{v : \tau};\ \texttt{)} \to (\underline{\sigma}, \otimes\underline{\pi};)\ \texttt{=}\ E(e', \otimes(\underline{w})),\\ \texttt{def}\ f\texttt{.lin(}\ u : \otimes\underline{\pi}; \underline{\dot{v} : \dot{\tau}}\ \texttt{)} \to (;\underline{\dot{\sigma}})\ \texttt{=}\ \texttt{let(}\ \otimes \underline{w} : \otimes\underline{\pi}\ \texttt{)} = u\ \texttt{in}\ \dot{e}'\end{array}}\ \textsc{UDef}$$

Fig. 8. Unzipping converting Linear A to Linear B. Unzipping transforms a mixed linear and non-linear expression e into a shared non-linear context E, a non-linear result expression e', and a linear result expression $\dot{e}'$. Note that we always unzip recursively, because Linear B does not allow non-linear expressions to read linear variables, even to drop them. Most language primitives are handled by the generic rules ULeaf and ULinLeaf.

Figure 8 gives the transformation $\mathcal{U}$[6] that "unzips" any Linear A expression or function into a pair of Linear B expressions or functions—one that computes only linear results and one that computes only non-linear results. Figure 9 shows the unzip of an example function.

The result of $\mathcal{U}$ on an expression is a bit subtle: we wish for $\mathcal{U}$ to split the input expression e into a non-linear fragment e' and a linear fragment $\dot{e}'$. However, e may compute a (non-linear) intermediate value that will be used for both linear and non-linear results. To allow e' and $\dot{e}'$ to both use that value without having to recompute it, $\mathcal{U}(e)$ also produces a context E of let bindings, which represent all the intermediate (non-linear) computations that e' and $\dot{e}'$ may share. All the actual work of the non-linear fragment ends up in the context E. The rule UDef exposes the variables

[6] $\mathcal{U}$ is not surfaced as a public API in JAX, but it is used internally by the AD implementation. The relevant implementation can be found in the file `jax/interpreters/partial_eval.py`.

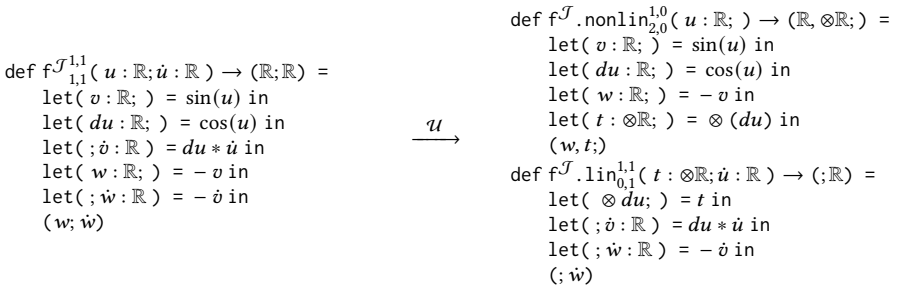


Fig. 9. Example of unzipping. On the left we have the derivative $\mathcal{J}(f)$ of $f(u) = -\sin(u)$ from Figure 6, and on the right the non-linear and linear functions it unzips to, $\mathcal{J}(f)$.nonlin and $\mathcal{J}(f)$.lin. In AD terminology, $\mathcal{J}(f)$.nonlin is the forward phase, which computes the original function f and returns all the intermediates, or the *tape*, (in this case the variable du) needed for the derivative. The function $\mathcal{J}(f)$.lin is called the linearization in Frostig et al. [2021]. It consumes the tape produced by the forward phase and computes (in substructurally and therefore algebraically linear fashion) the forward derivative of f; in Figure 10 (Section 7) we will transpose it to obtain the reverse phase for computing the derivative of f in reverse mode.

bound by E as outputs of the emitted non-linear fragment f.nonlin, so they can be read without recomputation by the linear fragment f.lin.

The context E is the "tape" (as it is often called [Griewank and Walther 2008]) that gives reverse-mode AD its work-preservation. In the example of Figure 9 it consists of the single variable dx. One of our contributions is disentangling the construction of the tape from AD proper. The time-space tradeoff the tape represents is handled by the unzipping transformation, and neither (forward) differentiation nor transposition need concern themselves with it.

6.1 Relation to Partial Evaluation

Unzipping can be seen as a form of partial evaluation (PE). Specifically, $\mathcal{U}$ symbolically partially evaluates f with respect to its non-linear arguments, leaving f.lin as the residual. The first difference with traditional PE [Jones et al. 1993] is that we do not require concrete values for the non-linear arguments of f. This is because our goal for unzipping is not to use static information to make the residual faster, but simply to separate the linear fragment so it can be transposed.

The second difference is that $\mathcal{U}$ always produces exactly two specializations of any function definition, whereas general partial evaluation may produce arbitrarily many. This is because we are always unzipping exactly the linear from the non-linear variables of every function, and Linear A does not permit the same function definition to be used with different arguments being treated linearly or non-linearly (even if the underlying computation is mathematically linear in different sets of its inputs).

We hope this identification of unzipping with partial evaluation may spur further advances in both PE and AD. In particular, checkpointing strategies developed for memory efficiency by the AD community might prove useful for general-purpose partial evaluation in memory-limited settings. In our system, checkpointing materializes thus: our ULet rule remains correct (though no longer work-preserving) if modified not to save a non-linear intermediate, and recompute it in the linear expression instead. Our transposition transformation $\mathcal{T}$ (Section 7) will then retain the non-linear recomputation while reversing the direction of the linear computation, implementing checkpointing. Perhaps similar ideas can be applied to partial evaluation more broadly.

6.2 Formalization

Given an expression e well-typed in $\Gamma; \dot{\Gamma}$ in Linear A, our transformation $\mathcal{U}(e)$ produces a context E (syntax in Figure 3), a non-linear expression e', and a linear *residual* expression $\dot{e}'$. We need the free non-linear variables $\underline{w}$ of the expression $\dot{e}'$, with their types $\underline{\pi}$. Then we define

- The *tape* $\underline{w}$. These are variables carrying the needed non-linear information from Γ and E to $\dot{e}'$.
- The *non-linear partial* $E(e', \underline{w})$. This is the expression formed by allocating fresh variables $\underline{u}$ for the results of e' and plugging $\texttt{let(}\ \underline{u};\ \texttt{)} = e'\ \texttt{in}\ (\underline{u}, \underline{w};)$ into the hole in E. The non-linear partial is a purely non-linear expression that returns the results of e' together with the tape.
- The *reconstruction* $E(e', \underline{w}; \dot{e}')$. Given fresh variables $\underline{u}$ and $\dot{\underline{u}}$ for all the results of e, this is the expression $\texttt{let(}\ \underline{u}, \underline{w};\ \texttt{)} = E(e', \underline{w})\ \texttt{in let(}\ ; \dot{\underline{u}}\ \texttt{)} = \dot{e}'\ \texttt{in}\ (\underline{u}; \dot{\underline{u}})$ that should be equivalent to e if the non-linear partial and the residual are correct. Note that here we bind $\underline{w}$ to themselves: the point is that we arrange for the context E not to be in scope in $\dot{e}'$, and carry all the needed information explicitly in the tape $\underline{w}$.

Theorem 6.2. ***Unzipping is total, correct, and work-preserving.*** *Consider any Linear A expression e well-typed in $\Gamma; \dot{\Gamma}$ with $\underline{w}$ and $\underline{\pi}$ as above. Then*

1. *$E; e'; \dot{e}' = \mathcal{U}(e)$ exist;*
2. *The non-linear partial $E(e', \underline{w})$ is well-typed in $\Gamma; \{\}$ in Linear B;*
3. *The residual $\dot{e}'$ is well-typed in $\underline{w} : \underline{\pi}; \dot{\Gamma}$ in Linear B; and*
4. *The reconstruction $E(e', \underline{w}; \dot{e}')$ is well-typed in $\Gamma; \dot{\Gamma}$ in Linear A and computes the same result with the same work as e.*

Proof. The proof consists of our usual structural induction. One subtlety is well-typedness of the non-linear partial $E(e', \underline{w})$ when e is a mixed `let` expression (rule ULet). We can hoist E_2 and e'_2 past the linear variables $\underline{v}$ because recursive unzipping will ensure that E_2 and e'_2 do not read any linear variables. We also need to check that the non-linear partial uses every non-linear variable at least once. By assumption, any non-linear variable v from Γ appears in either e_1 or e_2. By induction, v therefore appears in either the non-linear partial of e_1 or of e_2. If v appears in E_1, e'_1, E_2, or e'_2 directly, then v appears in either E or e'. Otherwise, v must either appear in the tape $\underline{w}_1$ due to being free in $\dot{e}'_1$, or in $\underline{w}_2$ due to being free in $\dot{e}'_2$. Therefore, v is free in $\dot{e}'$, because the latter only binds $\underline{\dot{v}}$, which cannot shadow v, being linear variables. Hence we will add v to the $\underline{w}$, and it will again appear in the non-linear partial $E(e, \underline{w})$ of e. A similar argument covers the non-linear variables $\underline{v}$ bound by the `let`. Checking variable consumption for the residual is a simpler variation of the same argument.

Some notes: The UDef rule coordinates with the UApp rule to transport the tape in a single variable of tuple type rather than increasing the size of every call site by the number of intermediate values of the function. The UApp rule relies on program ordering, because we read the type of the unzipped callee f`.nonlin` in order to annotate the type of the call site.[7] To enforce the invariant that non-linear expressions of Linear B do not consume linear variables, we explicitly unzip drop expressions in rule UDrop. The proof of claim 4 requires induction on the program to assert that any functions called by e compute the correct values, but is otherwise straightforward. □

[7] This is trivial for Linear A as presented, because it does not admit recursion; but this point becomes more subtle if recursion is permitted. Unzipping in a recursive language must be able to construct recursive types to represent the tapes of recursive computations. This can be accomplished by indirection: for each function f, allocate a fresh name π_f for the type of the tuple carrying the tape. Then unzipping can use that name fill in type annotations of call sites of f before f is unzipped, and unzipping f provides the definition of π_f (which may refer to π_f if f is recursive).

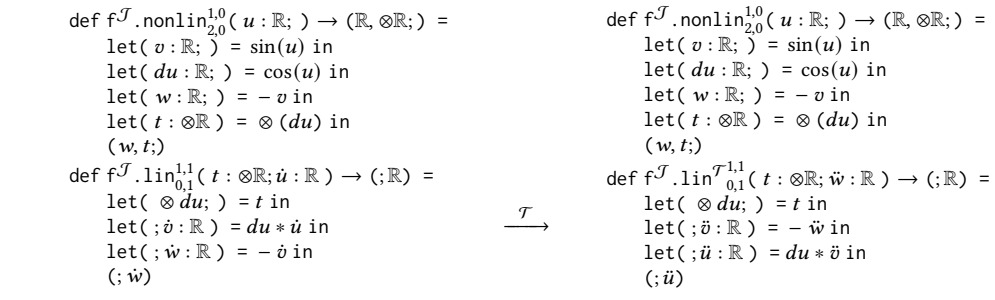


Fig. 10. Completing our running example by transposing the derivative. On the left, we have the forward phase $\mathcal{J}(f)$.nonlin and residual $\mathcal{J}(f)$.lin of $f(u) = -\sin(u)$ from Figure 9. On the right, we transpose just the linear residual $\mathcal{J}(f)$.lin to obtain the reverse phase $\mathcal{T}(\mathcal{J}(f)$.lin$)$, which computes the derivative of f in reverse mode.

7 TRANSPOSITION

To JVP, or VJP? That is the question:
Whether 'tis nobler in the mind to suffer
The slings and arrows of outrageous re-implementation,
Or to take up Arms against a Sea of troubles,
And by transposing, end them.

We turn to our main focus: transposing linear functions. Recall that our ultimate goal is to derive reverse-mode automatic differentiation from forward mode, without having to duplicate derivative rule implementations. We do this through the notion of transposition. Every linear function $f : \mathbb{R}^n \to \mathbb{R}^m$ has a unique transpose $f^{\mathcal{T}} : \mathbb{R}^m \to \mathbb{R}^n$ defined by consistency with respect to dot product:

$$\langle x, f^{\mathcal{T}}(y)\rangle = \langle f(x), y\rangle. \tag{1}$$

For a given nonlinear function $F : \mathbb{R}^n \to \mathbb{R}^m$ and input $x \in \mathbb{R}^n$, this is exactly the relationship between the forward derivative $dF : (y \in \mathbb{R}^n) \mapsto (JF|_x)y$ and the reverse derivative $dF^{\mathcal{T}} : (z \in \mathbb{R}^m) \mapsto z(JF|_x)$, where $JF|_x$ is the Jacobian of F at x.

Concretely, to compute a gradient of $F : \mathbb{R}^n \to \mathbb{R}$ at $\underline{x} \in \mathbb{R}^n$, we differentiate F with $\mathcal{J}$ to produce $F^{\mathcal{J}}$, unzip it into the linear and non-linear components $F^{\mathcal{J}}$.nonlin, $F^{\mathcal{J}}$.lin, transpose the latter to $F^{\mathcal{J}}$.lin$^{\mathcal{T}}$, and supply $1 \in \mathbb{R}$ as an initial gradient to the linear erasure thereof:

$$\nabla F|_{\underline{x}} = \left(\texttt{let(}\ \underline{y}; \underline{dx}\ \texttt{)} = F^{\mathcal{J}}\texttt{.nonlin(}\underline{x};\texttt{)}\ \texttt{in}\ \mathcal{L}(F^{\mathcal{J}}\texttt{.lin}^{\mathcal{T}})(\underline{dx}, 1;)\right).$$

An example transposition is given in Figure 10. We use the double-dot notation $\ddot{v}$ as a mnemonic for cotangent variables (not second-order derivatives). Each cotangent variable holds a partial derivative of the final result of F with respect to something, which are propagated in the opposite order from the primal computation. We assign no formal meaning to the dots themselves: $\ddot{v}$ is just a different variable than $\dot{v}$, even though we mean for it to carry the cotangent corresponding to $\dot{v}$.

7.1 Details

7.1.1 Input to Transposition. We transpose a function that's well-typed in Linear B with the $\mathcal{T}$ transformation[8] given in Figure 11. Recall from Definition 6.1 that Linear B is the sub-language of Linear A where all expressions must return either all-linear or all-non-linear results. We restrict to Linear B because general Linear A functions are free to interleave linear with non-linear computations, but we transpose only the strictly linear fragment. Converting an arbitrary Linear A expression into Linear B was the purpose of the unzipping transformation $\mathcal{U}$ of Section 6.

7.1.2 Transposition Environments. To read the transposition rules, it helps to view an expression as a function from its environment to its results, and to think of its environment as ordered. So, if $\dot{e}$ consumes the linear variables of an ordered type environment $\dot{\Delta}$, the transpose $\ddot{e}$ must produce one result corresponding to each variable in $\dot{\Delta}$. Likewise, for every result of $\dot{e}$, $\ddot{e}$ must consume a corresponding free variable. The transform $\mathcal{T}$ carries names for these (cotangent) variables in a second ordered type environment $\ddot{\Delta}$.

We use the notation $\dot{\Delta}$, $\ddot{\Delta}$ and angle brackets to emphasize that they are ordered vectors instead of the unordered mappings $\dot{\Gamma}$, $\ddot{\Gamma}$, and match the order of the relevant expression results. When we compose and decompose expressions and their corresponding ordered environments, it may be necessary to insert shuffling operations to maintain this ordering invariant. These shuffles are uninteresting and expressible using only let and multi-value return, both of which cost 0 in our model, so we leave them implicit. The ordering is irrelevant for type-checking, so we freely use $\dot{\Delta}$ and $\ddot{\Delta}$ as typing environments as well.

7.1.3 Omitted Rules. Figure 11 omits the rules TUnpack and TLinUnpack for non-linear and linear tuple unpacking, because they are identical to the corresponding multiple-value let rules, mutatis mutandis. There are no specific rules for transposing an expression that produces non-linear results because it transposes to itself (rule TNonLin). Linear B requires non-linear expressions to have no linear sub-expressions (not even through drop), so transposition need not traverse them.

7.1.4 Benefit from Explicit drop *and* dup*.* Now we have the payoff from requiring all linear variables to be explicitly consumed: without that, $\mathcal{T}(\dot{e})$ would have to return zeros for variables in $\dot{\Delta}$ that were not free variables of $\dot{e}$, and would later have to do work to add those zeros to the real cotangents. The payoff to requiring linear variables to be consumed exactly once is that rules do not need to introduce additions to merge the results of transposed sub-expressions—those cotangents are already guaranteed to correspond to different variables, and the summations are produced by the TDup rule.

7.2 Formalization

The goal of transposition is to transform a linear function f into its transpose $f^{\mathcal{T}}$, as defined by pulling back the dot product (1). We now prove that it does so, and that $f^{\mathcal{T}}$ isn't more expensive than f.

Theorem 7.1. ***Transposition is total, correct, and amortized work-preserving.*** *Consider a linear Linear B expression $\dot{e}$, well typed in non-linear and linear environments Γ, $\dot{\Gamma}$. Fix an ordering of variables in $\dot{\Gamma}$ to form $\dot{\Delta}$. Let $\underline{x}$ and $\underline{\dot{x}}$ be well-typed values corresponding to the free non-linear and linear variables in $\dot{e}$, respectively. Let $\underline{\ddot{x}}$ be values matching the types of the (linear) results from*

[8] $\mathcal{T}$ is available in JAX as the `jax.linear_transpose` function. Note that JAX does not distinguish between linear and non-linear expressions in its language, so the relevant typing judgements are reconstructed on the fly during transposition. The transform assumes that the given function can be typed according to the type system presented here and any failure to infer a valid type is reported to the user as an error.

$$\boxed{\mathcal{T}(\dot{\Delta}, \ddot{\Delta} \mid \dot{e}) \rightsquigarrow \ddot{e}}$$

$$\frac{}{\mathcal{T}(\langle \underline{\dot{v} : \dot{\tau}} \rangle, \langle \underline{\ddot{v} : \dot{\tau}} \rangle \mid (;\underline{\dot{v}})) \rightsquigarrow (;\underline{\ddot{v}})}\ \textsc{TRet}$$

$$\frac{\dot{\Delta}_2 = \langle \underline{\dot{w} : \dot{\sigma}} \rangle \quad \mathcal{T}(\langle \underline{\dot{w} : \dot{\sigma}}, \underline{\dot{v} : \dot{\tau}} \rangle, \ddot{\Delta} \mid \dot{e}_2) \rightsquigarrow \ddot{e}_2 \quad \mathcal{T}(\dot{\Delta}_1, \langle \underline{\ddot{v} : \dot{\tau}} \rangle \mid \dot{e}_1) \rightsquigarrow \ddot{e}_1 \quad \underline{\ddot{v}}, \underline{\ddot{w}} \text{ fresh}}{\mathcal{T}(\dot{\Delta}_1 \uplus \dot{\Delta}_2, \ddot{\Delta} \mid \texttt{let}(\ ;\underline{\dot{v} : \dot{\tau}}\ ) = \dot{e}_1 \texttt{ in } \dot{e}_2) \rightsquigarrow \texttt{let}(\ ;\underline{\ddot{w} : \dot{\sigma}}, \underline{\ddot{v} : \dot{\tau}}\ ) = \ddot{e}_2 \texttt{ in } (;\ddot{e}_1, \underline{\ddot{w}})}\ \textsc{TLetLin}$$

$$\frac{\mathcal{T}(\dot{\Delta}, \ddot{\Delta} \mid \dot{e}_2) \rightsquigarrow \ddot{e}_2 \quad \{\underline{v}\} \text{ non-empty}}{\mathcal{T}(\dot{\Delta}, \ddot{\Delta} \mid \texttt{let}(\ \underline{v : \tau};\ ) = e_1 \texttt{ in } \dot{e}_2) \rightsquigarrow \texttt{let}(\ \underline{v : \tau};\ ) = e_1 \texttt{ in } \ddot{e}_2}\ \textsc{TLetNonLin}$$

$$\frac{}{\mathcal{T}(\langle \underline{\dot{v} : \dot{\tau}} \rangle, \langle \underline{\ddot{v} : \ddot{\tau}} \rangle \mid f.\texttt{lin}(\underline{v}; \underline{\dot{v}})) \rightsquigarrow f.\texttt{lin}^{\mathcal{T}}(\underline{v}; \underline{\ddot{v}})}\ \textsc{TApp}$$

$$\frac{e \text{ non-linear}}{\mathcal{T}(\varepsilon, \varepsilon \mid e) \rightsquigarrow e}\ \textsc{TNonLin}$$

$$\frac{}{\mathcal{T}(\langle \dot{v} : \dot{\tau} \rangle, \langle \ddot{v} : \dot{\tau} \rangle \mid \dot{v}) \rightsquigarrow \ddot{v}}\ \textsc{TLinVar} \qquad \frac{}{\mathcal{T}(\epsilon, \langle \ddot{v} : \dot{\tau} \rangle \mid 0_{\dot{\tau}}) \rightsquigarrow \text{drop}(\ddot{v})}\ \textsc{TLinZero}$$

$$\frac{\underline{\ddot{w}} \text{ fresh}}{\mathcal{T}(\langle \underline{\dot{v} : \dot{\tau}} \rangle, \langle \ddot{v} : \otimes \underline{\dot{\tau}} \rangle \mid \otimes(\underline{\dot{v}})) \rightsquigarrow \texttt{let}(\ \otimes \underline{\ddot{w}} : \otimes \underline{\dot{\tau}}\ ) = \ddot{v} \texttt{ in } (;\underline{\ddot{w}})}\ \textsc{TLinTup}$$

$$\frac{}{\mathcal{T}(\langle \dot{v}_1 : \dot{\tau}, \dot{v}_2 : \dot{\tau} \rangle, \langle \ddot{v} : \dot{\tau} \rangle \mid \dot{v}_1 + \dot{v}_2) \rightsquigarrow \text{dup}(\ddot{v})}\ \textsc{TLinAdd}$$

$$\frac{}{\mathcal{T}(\langle \dot{v}_2 : \dot{\tau} \rangle, \langle \ddot{v}_2 : \dot{\tau} \rangle \mid v_1 * \dot{v}_2) \rightsquigarrow v_1 * \ddot{v}_2}\ \textsc{TLinMul}$$

$$\frac{}{\mathcal{T}(\langle \dot{v} : \dot{\tau} \rangle, \langle \ddot{v}_1 : \dot{\tau}, \ddot{v}_2 : \dot{\tau} \rangle \mid \text{dup}(\dot{v})) \rightsquigarrow \ddot{v}_1 + \ddot{v}_2}\ \textsc{TDup} \qquad \frac{\{\underline{\dot{v}}\} \text{ are the free variables of } \dot{e}}{\mathcal{T}(\langle \underline{\dot{v} : \dot{\tau}} \rangle, \epsilon \mid \text{drop}(\dot{e})) \rightsquigarrow (;0_{\dot{\tau}_i})}\ \textsc{TDropLin}$$

$$\boxed{\mathcal{T}(f.\texttt{lin}) \rightsquigarrow f.\texttt{lin}^{\mathcal{T}}}$$

$$\frac{\mathcal{T}(\langle \underline{\dot{v} : \dot{\tau}} \rangle, \langle \underline{\ddot{v} : \dot{\sigma}} \rangle \mid e) \rightsquigarrow e' \quad \underline{\ddot{v}} \text{ fresh}}{\mathcal{T}(\texttt{def } f.\texttt{lin}(\ \underline{v : \tau}; \underline{\dot{v} : \dot{\tau}}\ ) \to (;\underline{\dot{\sigma}}) = e) \rightsquigarrow \texttt{def } f.\texttt{lin}^{\mathcal{T}}(\ \underline{v : \tau}; \underline{\ddot{v} : \dot{\sigma}}\ ) \to (;\underline{\dot{\tau}}) = e'}\ \textsc{TDef}$$

Fig. 11. Linear B transposition. Linear B is the subset of Linear A where all functions and expressions return only linear or only non-linear results, and non-linear expressions do not read linear variables. Transposition leaves non-linear expressions as they are, but order-reverses linear expressions: a linear expression $\dot{e}$ with n free linear variables and m linear results becomes a linear expression $\ddot{e}$ with m free linear variables and n results computing the (linear-algebraic) transpose of $\dot{e}$.

$\dot{e}$, and let L_{in} and L_{out} be the number of scalars present in the linear inputs $\underline{\dot{x}}$ and (linear) results $\underline{\ddot{x}}$, respectively. Let $\mathcal{W}[\dot{e}]$ denote the amount of work done to execute $\dot{e}$. Let $\ddot{\Delta}$ be an ordered environment giving variable names and types for the cotangents $\underline{\ddot{x}}$ of the (linear) results of $\dot{e}$. Then

1. *$\ddot{e} = \mathcal{T}(\dot{\Delta}, \ddot{\Delta} | \dot{e})$ exists and is well-typed in Linear B, in environments $\Gamma; \ddot{\Delta}$, and emits linear results corresponding to $\dot{\Delta}$;*
2. *$\mathcal{W}[\ddot{e}] + L_{in} \leq \mathcal{W}[\dot{e}] + L_{out}$; and*
3. *$\langle \underline{\dot{x}}, \ddot{e}[\underline{x}; \underline{\ddot{x}}] \rangle = \langle \dot{e}[\underline{x}; \underline{\dot{x}}], \underline{\ddot{x}} \rangle$.*

Claim 2 is the amortized work preservation criterion for transposition. The amortization corrections L_{in} and L_{out} are the number of scalar linear inputs consumed by the original and transposed expressions, respectively. To see intuitively why a correction is needed, consider transposing a function that takes a single argument and fans it out (with dup) n times. In our cost model, this function does no work, because dup is free; but its transpose must perform n summations, and thus pay n. We restore work preservation by crediting an extra $(n + 1) - 1$ for the n excess inputs the transposition consumes.

Another intuition for the amortization argument is that producing L_{out} results obliges the caller to eventually pay L_{out} to drop or use them. A transposition that reduces that burden on the caller should get credit it can use to amortize the additional summations it has to do internally, and vice versa. These corrections cancel when Linear A expressions are composed, so transposition is exactly work-preserving internally, and the discrepancy only shows up for the end-to-end Linear A computation.

Claim 3 is the correctness criterion for transposition: $\ddot{e}$ pulls dot product in the output space of $\dot{e}$ back(ward) along $\dot{e}$ to its input space, which is exactly the mathematical definition of transposition of linear operators.[9] Note that on the left-hand side of Claim 3, the $\underline{\dot{x}}$ are treated as extra-linguistic values and $\underline{\ddot{x}}$ are fed in as linear inputs to $\ddot{e}$, whereas on the right-hand side, the $\underline{\ddot{x}}$ are extra-linguistic and the $\underline{\dot{x}}$ are fed in as linear inputs to $\dot{e}$.

While we state our results on transposition in terms of vector spaces over the familiar field $\mathbb{R}$ of real numbers, one can straightforwardly extend them to a variant of Linear A modeling modules over any ring $\mathcal{R}$. If $\mathcal{R}$ is not commutative, one needs to introduce a second linear scaling primitive $\dot{e} \cdot e$, corresponding to scalar right-multiplication in $\mathcal{R}$-modules. Scalar left- and right-multiplication then transpose to each other. Our dot-product notation also becomes non-commutative, and Claim 3 becomes two claims: $\langle \underline{\dot{x}}, \ddot{e}[\underline{x}; \underline{\dot{x}}] \rangle = \langle \dot{e}[\underline{x}; \underline{\dot{x}}], \underline{\ddot{x}} \rangle$ and $\langle \ddot{e}[\underline{x}; \underline{\ddot{x}}], \underline{\dot{x}} \rangle = \langle \underline{\ddot{x}}, \dot{e}[\underline{x}; \underline{\dot{x}}] \rangle$. Augmenting $\mathcal{R}$ with sufficient structure to define differentiation, and adjusting forward differentiation to correctly compute derivatives, is left as an exercise for the enterprising reader.

Proof of Claim 1. Well-typing of transposition follows by structural induction on the syntax of Linear B. The most interesting rule is TLetLin. The expression $\dot{e}$ is well-typed in $\Gamma; \dot{\Delta}$ by assumption, so we can split the incoming environment $\dot{\Delta}$ into a distinct union of the variables $\dot{\Delta}_1$ free in $\dot{e}_1$ and $\dot{\Delta}_2$ free in $\dot{e}_2$ (except for the $\underline{v}$ bound in this `let` form). The results of the overall expression $\dot{e}$ are the same as those of the body $\dot{e}_2$, so we can transpose $\dot{e}_2$ with respect to the augmented linear environment $\dot{\Delta}_2 \uplus \langle \underline{v : \tau} \rangle$ (up to a permutation) and the same expected results $\ddot{\Delta}$. This gives us the transposed body $\ddot{e}_2$, which by induction is well-typed in $\Gamma; \ddot{\Delta}$, and produces results corresponding to $\dot{\Delta}_2 \uplus \langle \underline{v : \tau} \rangle$ (up to a permutation). We now need to split them into the $\underline{\ddot{v}}$ corresponding to the $\underline{\dot{v}}$, which will be read by $\ddot{e}_1$, and the $\underline{\ddot{w}}$ corresponding to $\dot{\Delta}_2$, which we will return from the result $\ddot{e}$.

We now have the arguments needed to transpose the bound expression $\dot{e}_1$: its linear environment $\dot{\Delta}_1$ and variables $\underline{\ddot{v}}$ corresponding to its results. They line up correctly because the original expression $\dot{e}$ was well-typed as a `let`. Transposing $\dot{e}_1$ gives us $\ddot{e}_1$, which reads the $\underline{\ddot{v}}$ from its environment and produces results corresponding to $\dot{\Delta}_1$. Our result expression $\ddot{e}$ binds the $\underline{\ddot{v}}$ from $\ddot{e}_2$ so $\ddot{e}_1$ can read them, and splices the results corresponding to $\dot{\Delta}_1$ and $\dot{\Delta}_2$ into results corresponding to $\dot{\Delta}$. The result $\ddot{e}$ uses all the variables from $\ddot{\Delta}$ exactly once because the body $\ddot{e}_2$ does.

The other rules proceed similarly. One subtlety worth noting is that in TLetNonLin, we assume that all free linear variables of $\dot{e}$ are in fact free in the body $\dot{e}_2$, and none occur in the bound expression e_1. This rule is why Linear B expressly forbids non-linear expressions from consuming linear variables, which they might otherwise do with drop. □

[9] In the language of algebraic geometry, if $\dot{e}$ implements a pushforward of e, then $\ddot{e}$ implements a pullback of e.

Proof of Claim 2. Work preservation needs induction on the program as well as the syntax of expressions. The proof amounts to checking that the work corrections do, indeed, cancel. We again consider TLetLin first. If we denote by $|\cdot|$ the number of linear scalars present in some set of variables, we compute

$$\begin{aligned}
\mathcal{W}[\dot{e}] &= \mathcal{W}[\dot{e}_1] + \mathcal{W}[\dot{e}_2] && \text{cost of } \texttt{let}\\
&\geq \mathcal{W}[\dot{e}_1] + \mathcal{W}[\ddot{e}_2] + (|\dot{\Delta}_2| + |\underline{\dot{v}}|) - |\ddot{\Delta}| && \text{induction on } \dot{e}_2\\
&\geq \mathcal{W}[\ddot{e}_1] + |\dot{\Delta}_1| - |\underline{\ddot{v}}| + \mathcal{W}[\ddot{e}_2] + (|\dot{\Delta}_2| + |\underline{\dot{v}}|) - |\ddot{\Delta}| && \text{induction on } \dot{e}_1\\
&= \mathcal{W}[\ddot{e}] + |\dot{\Delta}| - |\ddot{\Delta}|, && \ddot{v}_i \text{ correspond to } \dot{v}_i
\end{aligned}$$

as desired.

Handling TApp is where we need to use induction on the program being transposed. To wit, in this case, $\mathcal{W}[\dot{e}]$ equals the work w done to evaluate (the body of) the function $f\,.\,\texttt{lin}$ being called, and $\mathcal{W}[\ddot{e}]$ equals the work $w^{\mathcal{T}}$ of $f\,.\,\texttt{lin}^{\mathcal{T}}$, and the corrections correspond as well. An induction only on expression syntax would not give us the needed premise that $w \geq w^{\mathcal{T}}$ + correction, but induction on the program does, because Linear B has no recursion.

We also find it instructive to attend to work preservation of TDup. The term $\mathrm{dup}(\dot{v})$ itself does no work in our cost model, but the addition it transposes to does work equal to $|\dot{v}|$, the number of real scalars in the variable. This is what we needed those corrections for: $\mathrm{dup}(\dot{v})$ produces $2|\dot{v}|$ results, while the addition only produces $|\dot{v}|$ results, so the corrected work is conserved. Similar reasoning applies to TAdd and TLit.

The rule TDrop is where the inequality in work preservation may be strict: we short-circuit the subexpression e, just emitting zero cotangents for its free linear variables. The resulting work $\mathcal{W}[0_{\mathcal{T}(\tau_i)}]$ is zero, corrected to $|\underline{\dot{v}}|$. To argue work preservation, we must invoke Lemma 4.3 to conclude that $\mathcal{W}[\mathrm{drop}(\dot{e})] \geq |\underline{\dot{v}}|$, and thus lower-bound the corrected work of the pre-transposition expression at $|\underline{\dot{v}}|$ as well. □

Proof of Claim 3. Correctness is also an induction on the program and the syntax of expressions. To give a flavor for the leaf-node behavior of transposition, let's prove correctness of TLinAdd and TDup. For type-compatible $\dot{v}_1, \dot{v}_2$, and $\ddot{v}$,

$$\langle (\dot{v}_1, \dot{v}_2), \mathrm{dup}(\ddot{v}) \rangle = \langle \dot{v}_1, \ddot{v} \rangle + \langle \dot{v}_2, \ddot{v} \rangle = \langle \dot{v}_1 + \dot{v}_2, \ddot{v} \rangle.$$

Read from left to right, this is exactly the correctness equation for TDup, and from right to left for TLinAdd. Note that this simple proof critically depends on the distributivity of multiplication over addition, also known as the linear factoring rule, which is crucial for the efficiency of reverse-mode autodiff (see e.g. Brunel et al. [2019]).

Transposition on interior AST nodes is exemplified by TLetLin. To begin the calculation, we stipulate the disjoint partition of $\underline{\dot{x}}$ into $\underline{\dot{x}}_1$ corresponding to $\dot{\Delta}_1$ and $\underline{\dot{x}}_2$ corresponding to $\dot{\Delta}_2$; and a similar (but not necessarily disjoint) split of $\underline{x}$ into $\underline{x}_1$ and $\underline{x}_2$. To simplify notation, we name the intermediate quantities $\underline{\dot{z}}$, $\underline{\ddot{z}}$, and $\underline{\ddot{x}}_2$, as, respectively, the forward evaluation of the bound expression $\dot{e}_1$, the corresponding backward evaluation of the transposed body $\ddot{e}_2$, and the residual results of $\ddot{e}_2$ which correspond to the free linear values $\underline{\dot{x}}_2$:

$$\begin{aligned}
\underline{\dot{z}} &= \dot{e}_1[\underline{x}_1; \underline{\dot{x}}_1]\\
\underline{\ddot{z}} \uplus \underline{\ddot{x}}_2 &= \ddot{e}_2[\underline{x}_2; \underline{\ddot{x}}].
\end{aligned}$$

Then we conclude the correctness proof for TLetLin by calculation:

$$\begin{aligned}
\langle \dot{e}[\underline{x}; \dot{\underline{x}}], \ddot{\underline{x}} \rangle &= \langle \dot{e}_2[\underline{x}_2; \dot{\underline{z}} \uplus \dot{\underline{x}}_2], \ddot{\underline{x}} \rangle && \text{1-step evaluation of } \dot{e} \\
&= \langle \dot{\underline{z}} \uplus \dot{\underline{x}}_2, \ddot{e}_2[\underline{x}_2; \ddot{\underline{x}}] \rangle && \text{induction on } \dot{e}_2 \\
&= \langle \dot{\underline{z}}, \ddot{\underline{z}} \rangle + \langle \dot{\underline{x}}_2, \ddot{\underline{x}}_2 \rangle && \text{partitioning dot product} \\
&= \langle \dot{\underline{x}}_1, \ddot{e}_1[\underline{x}_1; \ddot{\underline{z}}] \rangle + \langle \dot{\underline{x}}_2, \ddot{\underline{x}}_2 \rangle && \text{induction on } \dot{e}_1 \\
&= \langle \dot{\underline{x}}, \ddot{e}_1[\underline{x}_1; \ddot{\underline{z}}] \uplus \ddot{\underline{x}}_2 \rangle && \text{unpartitioning dot product} \\
&= \langle \dot{\underline{x}}, \ddot{e}[\underline{x}; \ddot{\underline{x}}] \rangle. && \text{1-step (un)evaluation of } \ddot{e}
\end{aligned}$$

Other syntactic forms proceed similarly, and more simply. TApp again requires induction on the whole program being transposed. □

Corollary 7.2. *If we transpose an expression $\dot{e}$ twice, the result is equivalent to $\dot{e}$ and does at most the same amount of work.*

8 CODE SIZE

When implementing practical compilers, the size of the (intermediate) program representation is also an important consideration alongside its ultimate runtime, because it limits the runtime and memory cost of the compiler itself. We thus emphasize that our transformations increase code size only linearly, because they turn function definitions and function calls into new definitions and calls, without duplicating or inlining function bodies.

We can argue this formally by modeling the size $|P|$ of a Linear A program P simply as "every node in the grammar from Figure 3 has size 1". We then deduce

Theorem 8.1. ***Erasure, differentiation, unzipping, and transposition are size-preserving.*** *There exist constants $c_{\mathcal{L}}$, $c_{\mathcal{T}}$, $c_{\mathcal{U}}$, $c_{\mathcal{J}}$ such that for any Linear A program P,*

$$\begin{aligned}
|\mathcal{L}(P)| &\leq c_{\mathcal{L}}|P|, \\
|\mathcal{J}(P)| &\leq c_{\mathcal{J}}|P|, \\
|\mathcal{U}(P)| &\leq c_{\mathcal{U}}|P|, \text{ and} \\
|\mathcal{T}(P)| &\leq c_{\mathcal{T}}|P|,
\end{aligned}$$

where $\mathcal{L}$, $\mathcal{J}$, $\mathcal{U}$, and $\mathcal{T}$ are the erasure, differentiation, unzipping, and transposition transformations defined in Sections 4.5, 5, 6, and 7, respectively.

Proof. Mostly by inspection, but there is one subtlety. In rule UApp (Section 6), the type σ of the tape of the applied function f appears in the emitted let binding. In Linear A as presented, σ will have size proportional to the number of non-linear intermediates in f, which may be asymptotically larger than the call site being transformed. This is, however, easy to fix: just augment the type system with synonyms, and annotate that binding with a synonym for the tape instead. When function calls are nested, this amounts to retaining structure sharing in the tape types.

The same phenomenon is also why linearity erasure $\mathcal{L}$ (Section 4.5) has to introduce dedicated functions for vector space operations on arbitrary types, instead of inlining them. □

The constant $c_{\mathcal{J}}$ depends on the set of primitives and their derivatives defined for Linear A, but $c_{\mathcal{L}}$, $c_{\mathcal{U}}$ and $c_{\mathcal{T}}$ are absolute, given by the size of the expressions constructed by our rules for $\mathcal{L}$, $\mathcal{U}$ and $\mathcal{T}$, respectively. It is necessary to count variable references in program size (even though we model them with zero runtime cost), because for instance $\mathcal{J}$ introduces calls to dup for repeated references to (non-linear) variables.

9 RELATED WORK

Most of the material presented here can be found in one form or another in existing work, but the composition of individual pieces is novel. The AD architecture we describe was sketched out briefly in Frostig et al. [2021] and again in Paszke et al. [2021], and is in use in the JAX and Dex projects [Bradbury et al. 2018; Frostig et al. 2018]. Transposition as a program transformation has been previously explored by Piponi [2009]. Unzipping is an instance of partial evaluation—a technique widely known and explored [Futamura 1983; Jones et al. 1993]—with the difference that we are interested not only in the residual program, but also in the program formed by operations that are considered "statically computable". Automatic differentiation itself has seen rapid growth in recent years, especially for functional languages, which we attempt to briefly summarize below.

Forward-mode differentiation is significantly simpler to embed in existing programming languages, for example by the means of templates [Piponi 2004], operator overloading [Walther et al. 2003] and Haskell-style typeclasses or ML-style functors [Elliott 2009; Karczmarczuk 1998]. The true difficulty, both conceptually and implementation-wise, arises for reverse mode.

Initially, reverse-mode autodiff development was focused on imperative languages. In particular, the implicit reuse of values in the original computation was consistently translated to destructive mutation. This made higher-order differentiation more difficult, and meant that it generally did not have a natural non-monadic translation into pure functional languages. This challenge was overcome by Pearlmutter and Siskind [2008b], with the introduction of reverse mode as an augmentation of the original program with "backpropagator" closures replacing the tape. As shown by Wang et al. [2019] a similar transformation can be performed by expressing the non-local control using delimited continuations, manifested by the shift and reset operators. Furthermore, Brunel et al. [2019] use a similar method to perform provably correct reverse-mode differentiation over lambda calculus extended with linear negation. Especially interestingly, they already make the connection between linear logic and algebraic linearity that underlies the work presented here. Finally, Krawiec et al. [2022] outline a provably correct and efficient implementation of AD in a higher-order language, at a cost of requiring an extra-linguistic monadic translation.

The purpose of the monadic translation of Krawiec et al. [2022] is to stage out a program, expressing the linearization of primal evaluation at a given point, as the primal is running. But, since control flow is dependent only on the non-linear values, the staged program never uses control flow or higher-order functions itself (linearization of only the branches taken gets inlined). In fact, it uses a language almost identical to the linear part of Linear A.

That approach can be factorized through the lens taken in our work. Krawiec et al. [2022] describes reverse-mode AD in two phases. The first phase simultaneously: traces the metalanguage into a representation without control flow and higher-order functions (our Linear A); performs forward mode AD (our $\mathcal{J}$); and partially evaluates (like our unzipping $\mathcal{U}$, but specializing the linear function to exact values, instead of separating non-linear compute). In the second phase, transposition is implemented as an evaluator function that traverses the staged linear program.

Complementing the previous implementation-focused work, multiple explorations of autodiff semantics have been developed. Many of those give up the asymptotic efficiency guarantees, which are critical for practical implementations of autodiff, but they do help shed light onto the underlying theory. Elliott [2018] outlines a minimal autodiff system for a language following category theoretical foundations. Similarly to our work he also extensively reasons about the linearity of derivatives, including the introduction of a linear function type. Abadi and Plotkin [2019] describe consistent operational and denotational semantics for autodiff in a first-order language, while Huot et al. [2020] extend it to a higher-order language using diffeology semantics.

Mazza and Pagani [2021] outline semantics for a higher-order language with conditionals that are consistent with the mathematical notion of differentiation everywhere except for a measure 0 set.

10 DISCUSSION AND FUTURE WORK

Extensibility. Decomposing reverse-mode AD can lead to fewer AD rules overall. Consider a language with N primitives, H of which are higher-order (think control flow) and L are linear. If the two AD modes are separate, then a system implementer has to provide $N + N$ rules—one for each primitive for each mode. However, in our case, $N+H+L$ suffice: one still implements forward-mode for all primitives, but unzipping acts over all first-order primitives uniformly, and transposition only deals with linear primitives. Many first-order array languages support tens or even hundreds of primitives, most of which are both first-order and non-linear, leading to significant savings.

As a corollary, with our approach, an AD system user who wants to customize AD behavior on a subprogram only needs to provide the custom forward-mode implementation, as the customized reverse-mode implementation can be derived for them automatically.

Control flow. Data-dependent control is the most glaring omission from Linear A. While Linear A remains useful as a model of the straightline computation results from tracing at a given primal point, adding "if" or "case" (and loops or recursive functions) would broaden it to a model of the whole surrounding numerical language.

Control would be a substantial extension: control flow means that different branches may compute different intermediate values, so one must also extend the type system to allow sum types (which were not necessary in Linear A). Having sum types, one must then concern oneself with defining the tangent type of a sum. It would be most satisfying to use a dependent type system, so as to preserve the invariant that only non-linear variables may have sum types, and linear computations operate only on products of $\mathbb{R}$. *Which* product, however, can depend on a previous non-linear computation. Such a type system would also capture the constraint that the value tangent to a non-linear sum must inhabit the same variant.

Multilinearity. Linear A does not model multilinearity. To wit, multiplication is actually linear separately in both arguments, whereas its model in our system treats the first argument non-linearly. More generally, a function like $f(x, y, z) \mapsto z(x + y)$ can be viewed as linear in z, or as linear in (x, y) jointly. Our linear languages force the user to choose one, or to duplicate the function if both interpretations are necessary; whereas it would be more satisfying to be able to encode all of those linearity properties in a type system at once.

Multilinearity seems to be an important ingredient in two other directions of generalization. First, a program containing independent derivatives will have components that are independently linear; which can be important to track, especially if those derivatives interact (e.g., if they are nested). This is important to address in order to give a satisfying, linearity-aware typing to in-language automatic differentiation (as opposed to the extra-linguistic transformations we dealt with here).

Second, multilinearity is part of extending these ideas to higher-order functions. When asking what it means for a function-returning function such as $f : \mathbb{R} \to (\mathbb{R} \to \mathbb{R})$ to be linear, a natural answer seems to be that the final result should be linear in both inputs separately, i.e., multilinear. However, any system covering higher-order functions should be able to give a linear type to curried addition as well, where the final result is linear in both arguments jointly.

Partial evaluation. We have observed that our unzipping transformation $\mathcal{U}$ is an instance of partial evaluation. Unzipping has a time-space tradeoff: for each non-linear intermediate, $\mathcal{U}$ can either store the result on the tape (to minimize work), or emit code in the linear residual that recomputes it (to reduce memory pressure by the emitted program). The AD community has devoted considerable attention to this tradeoff under the heading of checkpointing AD; perhaps there are ideas there that could be ported to general partial evaluation in memory-limited settings.

Array size polymorphism. One of the limitations of Linear A as defined is that the size of any tuple is syntactically apparent, so it is not possible to write size-polymorphic (i.e., "array processing") functions in Linear A. Array-size-polymorphic functions are of course critical in practice, but are mostly orthogonal to the transformations we introduced here. We leave a full formalization to future work, with the comment that JAX resolves array size polymorphism at trace time (in our terms, during metaprogramming of Linear A), whereas Dex handles it by extending the internal representation with constructs that deal with it directly.

Sparse matrix algorithms. In a different direction, indexed linear functions are a representation for matrices. Indeed, they are a maximally expressive representation for matrices, that can encode any structured (or unstructured) sparsity pattern whatever. This statement is not very deep: the sine qua non of a sparse matrix representation is to be able to compute matrix-vector products Mx; the function that closes over the matrix M and computes that product with a vector x it accepts as argument is algebraically linear; and it seems obvious that any such function can be written to type-check as linear in x in some reasonable variant of Linear A (perhaps extended with control flow as needed).

From this lens, our contribution is a universal sparsity-preserving transposition operation for sparse matrix representations (as well as a formalization of the intuition that an automatic derivative is a sparse representation of the Jacobian of the differentiated function). Are there any other sparse-matrix operations that can be rendered universal as code transformations on Linear A? Can sparse-matrix algorithms for, say, matrix multiplication be fruitfully recovered from a known code transformation (e.g., simplification or partial evaluation) applied to the composition of the corresponding indexed linear functions? Can new sparse-matrix algorithms or representations be derived more simply or robustly from this direction?

11 CONCLUSION

We presented a decomposition of reverse-mode automatic differentiation into three distinct program transformations: forward-mode differentiation, unzipping, and transposition. The interface between these transformations is a (substructural) linear type system for checking (algebraic) linearity. Automatic derivatives type-check as linear, unzipping separates the linear and non-linear parts of a computation, and transposition runs the linear part backward to efficiently compute its transpose.

This decomposition clarifies and simplifies automatic differentiation systems, by separating the mind-bending direction-reversal needed for reverse-mode AD from the actual differentiating. This decomposition also sheds light on checkpointing strategies—the decision of whether to save an intermediate value or recompute it is entirely the province of the unzipping transformation, which amounts to performing partial evaluation in a way that trades off run-time computation for lower memory usage. Transposition handles any set of choices mechanically.

12 ACKNOWLEDGEMENTS

The authors would like to thank Pavel Sountsov for the observation that Linear A does not handle shape polymorphism, Colin Carroll for the use case of simplifying custom gradients, and Gordon Plotkin for introducing us to linearity, especially the connection between linear algebra and linear types. We also thank our anonymous reviewers for their remarkably detailed and insightful feedback.

REFERENCES

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

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. Google. http://github.com/google/jax

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

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

Conal M. Elliott. 2009. Beautiful Differentiation. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming* (Edinburgh, Scotland) *(ICFP '09)*. Association for Computing Machinery, New York, NY, USA, 191–202. https://doi.org/10.1145/1596550.1596579

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

Roy Frostig, Matthew J. Johnson, Dougal Maclaurin, Adam Paszke, and Alexey Radul. 2021. Decomposing reverse-mode automatic differentiation. In *LAFI: POPL workshop on Languages for Inference*. Association for Computing Machinery, New York, NY, USA. https://arxiv.org/abs/2105.09469

Yoshihiko Futamura. 1983. Partial computation of programs. In *RIMS Symposia on Software Science and Engineering*, Eiichi Goto, Koichi Furukawa, Reiji Nakajima, Ikuo Nakata, and Akinori Yonezawa (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–35. https://doi.org/10.1007/3-540-11980-9_13

Andreas Griewank and Andrea Walther. 2008. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation* (second ed.). Society for Industrial and Applied Mathematics, USA. https://doi.org/10.1137/1.9780898717761

Mathieu Huot, Sam Staton, and Matthijs Vákár. 2020. Correctness of Automatic Differentiation via Diffeologies and Categorical Gluing. In *Foundations of Software Science and Computation Structures - 23rd International Conference, FOSSACS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings (Lecture Notes in Computer Science, Vol. 12077)*, Jean Goubault-Larrecq and Barbara König (Eds.). Springer, 319–338. https://doi.org/10.1007/978-3-030-45231-5_17

Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. 1993. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, Inc., USA.

Jerzy Karczmarczuk. 1998. Functional Differentiation of Computer Programs. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming* (Baltimore, Maryland, USA) *(ICFP '98)*. Association for Computing Machinery, New York, NY, USA, 195–203. https://doi.org/10.1145/289423.289442

Faustyna Krawiec, Simon Peyton Jones, Neel Krishnaswami, Tom Ellis, Richard A. Eisenberg, and Andrew Fitzgibbon. 2022. Provably Correct, Asymptotically Efficient, Higher-Order Reverse-Mode Automatic Differentiation. *Proc. ACM Program. Lang.* 6, POPL, Article 48 (jan 2022), 30 pages. https://doi.org/10.1145/3498710

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

Adam Paszke, Daniel D. Johnson, David Duvenaud, Dimitrios Vytiniotis, Alexey Radul, Matthew J. 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), 29 pages. https://doi.org/10.1145/3473593

Barak A Pearlmutter and Jeffrey Mark Siskind. 2008a. Reverse-mode AD in a functional framework: Lambda the ultimate backpropagator. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 30, 2 (2008), 1–36. https://doi.org/10.1145/1330017.1330018

Barak A. Pearlmutter and Jeffrey Mark Siskind. 2008b. Reverse-Mode AD in a Functional Framework: Lambda the Ultimate Backpropagator. *ACM Trans. Program. Lang. Syst.* 30, 2, Article 7 (mar 2008), 36 pages. https://doi.org/10.1145/1330017.1330018

Dan Piponi. 2004. Automatic Differentiation, C++ Templates, and Photogrammetry. *Journal of Graphics Tools* 9, 4 (2004), 41–55. https://doi.org/10.1080/10867651.2004.10504901

Dan Piponi. 2009. Two Tricks for the Price of One: Linear Filters and Their Transposes. *J. Graphics, GPU, & Game Tools* 14, 1 (2009), 63–72. https://doi.org/10.1080/2151237X.2009.10129275

Amr Sabry and Matthias Felleisen. 1992. Reasoning about Programs in Continuation-Passing Style. *SIGPLAN Lisp Pointers* V, 1 (jan 1992), 288–298. https://doi.org/10.1145/141478.141563

Andrea Walther, Andreas Griewank, and Olaf Vogel. 2003. ADOL-C: Automatic Differentiation Using Operator Overloading in C++. *PAMM* 2, 1 (2003), 41–44. https://doi.org/10.1002/pamm.200310011

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 (jul 2019), 31 pages. https://doi.org/10.1145/3341700

Received 2022-07-07; accepted 2022-11-07