Efficient Dual-Numbers Reverse AD via Well-Known Program Transformations*

TOM J. SMEDING, Utrecht University, The Netherlands
MATTHIJS I. L. VÁKÁR, Utrecht University, The Netherlands

Where dual-numbers forward-mode automatic differentiation (AD) pairs each scalar value with its tangent value, dual-numbers *reverse-mode* AD attempts to achieve reverse AD using a similarly simple idea: by pairing each scalar value with a backpropagator function. Its correctness and efficiency on higher-order input languages have been analysed by Brunel, Mazza and Pagani, but this analysis used a custom operational semantics for which it is unclear whether it can be implemented efficiently. We take inspiration from their use of *linear factoring* to optimise dual-numbers reverse-mode AD to an algorithm that has the correct complexity and enjoys an efficient implementation in a standard functional language with support for mutable arrays, such as Haskell. Aside from the linear factoring ingredient, our optimisation steps consist of well-known ideas from the functional programming community. We demonstrate the use of our technique by providing a practical implementation that differentiates most of Haskell98.

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

Additional Key Words and Phrases: automatic differentiation, source transformation, functional programming

ACM Reference Format:
Tom J. Smeding and Matthijs I. L. Vákár. 2023. Efficient Dual-Numbers Reverse AD via Well-Known Program Transformations. *Proc. ACM Program. Lang.* 7, POPL, Article 54 (January 2023), 28 pages. https://doi.org/10.1145/3571247

1 INTRODUCTION

An increasing number of applications requires computing derivatives of functions specified by a computer program. The derivative of a function gives more qualitative information of its behaviour around a point (i.e. the local shape of the function's graph) than just the function value at that point. This qualitative information is useful, for example, for optimising parameters along the function (because the derivative tells you how the function changes) or inferring statistics about the function (e.g. an approximation of its integral). These uses appear, respectively, in parameter optimisation in machine learning or numerical equation solving, and in Bayesian inference of probabilistic programs. Both application areas are highly relevant today.

Automatic differentiation (AD) is the most effective technique for efficient computation of derivatives of programs, and comes in two main flavours: forward AD and reverse AD. In practice, by far the most common case is that functions have many input parameters and few, or even only one, output parameters; in this situation, forward AD is inefficient while reverse AD yields

*This project has received funding via NWO Veni grant number VI.Veni.202.124.

Authors' addresses: Tom J. Smeding, Department of Information and Computing Sciences, Utrecht University, Utrecht, The Netherlands, t.j.smeding@uu.nl; Matthijs I. L. Vákár, Department of Information and Computing Sciences, Utrecht University, Utrecht, The Netherlands, m.i.l.vakar@uu.nl.

2475-1421/2023/1-ART54
https://doi.org/10.1145/3571247

the desired computational complexity. Indeed, reverse AD computes the gradient of a function implemented as a program in time at most a constant factor more than runtime of the original program, where forward AD has multiplicative overhead in the size of the program input. However, reverse AD is also significantly more difficult to implement flexibly and correctly than forward AD.

Many approaches exist for doing reverse AD on a higher-order language: using taping/tracing in an imperative language (e.g. [Paszke et al. 2017]) and in a functional language [Kmett and contributors 2021], using linearisation and transposition code transformations [Paszke et al. 2021], or sometimes specialised by taking advantage of common usage patterns in domain-specific languages [Schenck et al. 2022]. In the theory community, various algorithms have been described that apply to a wide variety of source languages, including approaches based on symbolic execution and tracing [Abadi and Plotkin 2020; Brunel et al. 2020] and on category theory [Vákár and Smeding 2022], as well as formalisations of existing implementations [Krawiec et al. 2022]. Despite the fact that all these source languages could, theoretically, be translated to a single generic higher-order functional language, each reverse AD algorithm takes a different approach to solve the same problem. It is unclear how exactly these algorithms relate to each other, meaning that correctness proofs (if any) need to be rewritten for each individual algorithm.

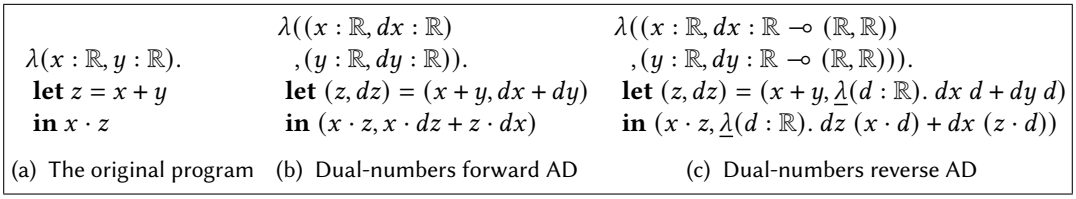
This paper aims to improve on the situation by providing a link from the elegant dual-numbers reverse AD algorithm analysed in [Brunel et al. 2020] to a functional taping approach as used in [Kmett and contributors 2021] and analysed in [Krawiec et al. 2022]. The key point made by Brunel, Mazza and Pagani [Brunel et al. 2020] is that one can attain the right computational complexity by starting from the very elegant dual-numbers reverse AD code transformation (Sections 2 and 3), and adding a *linear factoring* rule to the operational semantics of the output language of the code transformation. This linear factoring reduction rule states that for linear functions f, the expression $f\ x + f\ y$ should be reduced to $f\ (x + y)$.

Our main contributions are the following:

- We show how the theoretical analysis based on the linear factoring rule can be used as a basis for an algorithm that assumes normal, call-by-value semantics. We do this by *staging calls to backpropagators* in Section 4.
- We show how this algorithm can be made efficient by using the standard functional programming techniques of Cayley transformation (Section 5) and (e.g. linearly typed or monadic) functional in-place updates (Section 7).
- We explain how our algorithm relates to classical taping-based approaches (Section 8).
- We give an implementation of the final algorithm of Section 8.2 that can differentiate most of Haskell98 (but using call-by-value semantics), and that has the correct asymptotic complexity, as well as reasonable constant-factor performance (Section 10).
- We explain in detail how our technique relates to the functional taping AD of [Kmett and contributors 2021] and [Krawiec et al. 2022] as well as [Shaikhha et al. 2019]'s approach of trying to optimise forward AD to reverse AD at compile time (Section 12). We also briefly describe the broader relationship with related work.

2 KEY IDEAS

Naive dual-numbers reverse AD. In traditional dual-numbers *forward* AD, one pairs up the real scalars in the input of a program with their *tangent* (these tangents together form the *directional derivative* of the input), runs the program with overloaded arithmetic operators to propagate forward these tangents, and finally reads the tangent of the output from the tangents paired up with the output scalars of the program. For example, transforming the program in Fig. 1a using dual-numbers forward AD yields Fig. 1b.



(a) The original program (b) Dual-numbers forward AD (c) Dual-numbers reverse AD

Fig. 1. An example program together with its derivative, both using dual-numbers forward AD and using dual-numbers reverse AD. The original program is of type $(\mathbb{R}, \mathbb{R}) \to \mathbb{R}$.

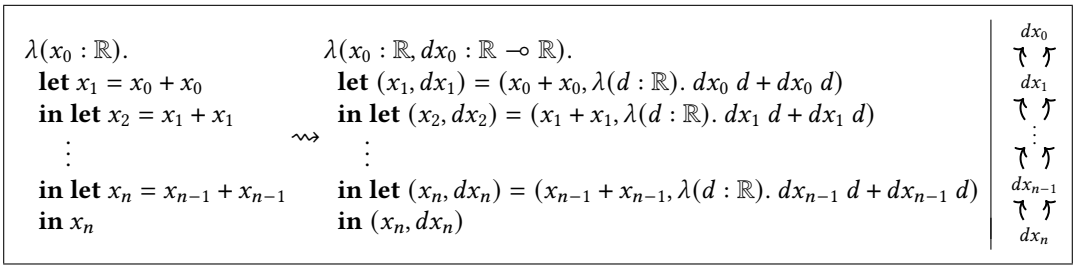


Fig. 2. Left: an example showing how naive dual-numbers reverse AD can result in exponential blow-up when applied to a program with sharing. Right: the dependency graph of the backpropagators dx_i.

For reverse AD, such an elegant formulation is also possible, but we have to somehow encode the "reversal" in the tangent scalars that we called dx and dy in Fig. 1b. A solution is to replace those tangent scalars with *linear functions* that take the *cotangent* (or *reverse derivative*, or *adjoint*) of the scalar it is paired with, and return the cotangent of the full input of the program. Transforming the same example program Fig. 1a using this style of reverse AD yields Fig. 1c. The linearity indicated by the $\multimap$-arrow here is that of a monoid homomorphism (a function preserving 0 and (+)); however, operationally, linear functions behave just like regular functions.

This naive dual-numbers reverse AD transformation, which we list in Fig. 6, is simple and it is easy to see that it is correct via a logical relations argument [Nunes and Vákár 2022b]. The idea of this argument is to prove via induction that a backpropagator $x' : \mathbb{R} \multimap c$ that is paired with an intermediate value $x : \mathbb{R}$ in the program, holds the gradient of the computation that calculates $x : \mathbb{R}$ from the global input of type c.

Dual-numbers forward AD has the very useful property that it generalises over many types (e.g. products, coproducts, recursive types) and program constructs (e.g. recursion, higher-order functions), thereby being applicable to e.g. all of Haskell98; the same property is inherited by the style of dual-numbers reverse AD exemplified here. However, unlike dual-numbers forward AD (which can propagate tangents through a program with only a constant-factor overhead over the original runtime), dual-numbers reverse AD is wildly inefficient: calling dx_n returned by the differentiated program in Fig. 2 takes time *exponential* in n. Such overhead would make reverse AD completely useless in practice—particularly because other (less flexible) reverse AD algorithms exist that indeed do a lot better. (See e.g. [Baydin et al. 2017].)

However, it turns out that this naive form of dual-numbers reverse AD can be *optimised* to be as efficient (in terms of time complexity) as these other algorithms—and most of these optimisations are just applications of standard functional programming techniques. This paper presents a sequence of changes to the code transformation (see the overview in Fig. 3) that fix all the complexity issues and, in the end, produce an algorithm with which the differentiated program has only a constant-factor overhead in runtime over the original program. This complexity is as desired from a reverse AD algorithm, and is best possible, while nevertheless being applicable to a wide range of programming language features. We explain how the result is essentially equivalent to classical taping techniques.

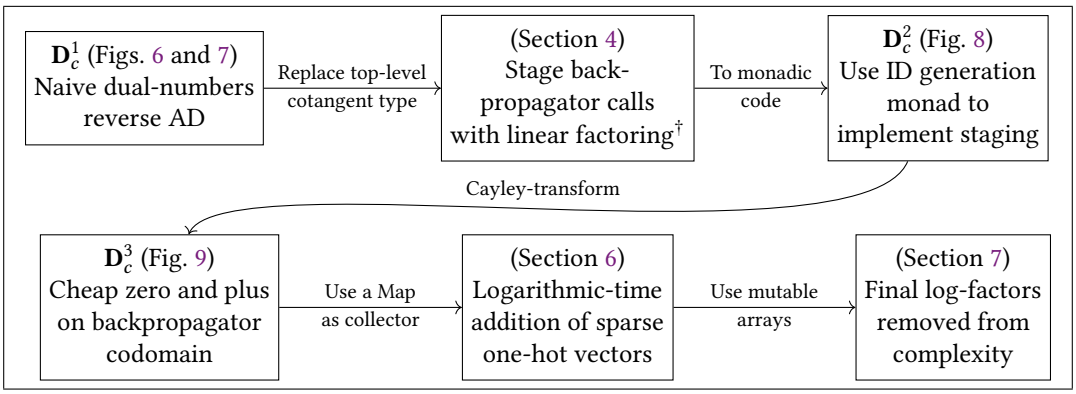


Fig. 3. Overview of the optimisations to dual-numbers reverse AD as a code transformation that are described in this paper. († = inspired by [Brunel et al. 2020])

Optimisation steps. We present, in Fig. 3, an overview of the optimisations that we apply to the dual-numbers reverse AD algorithm to fix its complexity problems. We discuss these one by one.

We first apply *linear factoring*: for a linear function f, such as a backpropagator, we have that $f\ x + f\ y = f\ (x+y)$. Observing the form of the backpropagators in Fig. 6, we see that in the end all we produce is a giant sum of applications of backpropagators to scalars; hence, in this giant sum, we should be able to contract applications to the same backpropagator using this linear factoring rule.

We achieve this linear factoring by not returning a plain c (presumably the type of the program input) from our backpropagators, but instead a c wrapped in an object that can delay calls to linear functions producing a c. This object we call Staged; aside from changing the monoid that we are mapping into from $(c, \underline{0}, (+))$ to $(\mathrm{Staged}\, c, 0_{\mathrm{Staged}}, (+_{\mathrm{Staged}}))$, the only material change is that the calls to d_i in $\mathbf{D}_c^1[\![op]\!]$ are now wrapped using a new function SCall, which delays the calls to d_i by storing the relevant metadata in the returned Staged object.

However, it is not obvious how to implement this Staged type: we need a linear order on (linear) function values if they are to serve as keys in a tree map, which the Staged interface currently prescribes. Furthermore, even if we can delay calls to backpropagators, we still need to call them at some point, and it is unclear in what order we should do so (and this order turns out to be very important). We solve these problems by generating, at runtime, a unique identifier (ID) for each backpropagator that we create, which we do by letting the differentiated program run in an ID generation monad (a special case of a state monad). The result is shown in Fig. 8, which is very similar to the previous version in Fig. 6 apart from threading through the next-ID-to-generate. (The code looks very different, but this is only due to monadic bookkeeping.)

At this point, the code transformation reaches a significant milestone: by staging (delaying) calls to backpropagators as long as possible, we can ensure that *every backpropagator is called at most once.* This milestone is achieved using the following observation: if we assign incrementing numeric IDs at runtime to lambda functions in a pure functional program, then the runtime closure of a lambda function can only refer to other functions with *smaller* IDs. Now, since our backpropagators only call other functions contained in their closure (and not functions contained in their input argument), we can use the observation to conclude that all backpropagator calls are to other backpropagators with smaller IDs! This allows us to resolve all backpropagator calls in the Staged object produced by the differentated program by simply calling them from highest ID to lowest ID, and collecting and combining (using linear factoring) the calls made along the way. (See Section 4.)

But we are not done yet. The code transformation at this point ($\mathbf{D}_c^2$ in Fig. 8) still has a glaring problem: orthogonal to the issue that backpropagators were called too many times (which we

fixed), we are still creating one-hot input cotangents and adding those together. This problem is somewhat more subtle, because it is not actually apparent in the program transformation itself; indeed, looking back at Fig. 1c, there are no one-hot values to be found. However, the only way to *use* the program in Fig. 1c to do something useful, namely to compute the cotangent (gradient) of the input, is to pass $(\lambda z.\ (z, 0))$ to dx and $(\lambda z.\ (0, z))$ to dy; it is easy to see that generalising this to larger input structures results in input values like $(0, \ldots, 0, z, 0, \ldots, 0)$ that get added together. Adding many zeros together can hardly be the most efficient way to go about things, and indeed this is a complexity issue in the algorithm.

The way we solve this problem of one-hots is less AD-specific: the most important optimisations that we perform are Cayley-transformation (Section 5) and using a better sparse vector representation (Map Int $\mathbb{R}$ instead of a plain c value; Section 6). Cayley-transformation (also known as *difference lists* [Hughes 1986] in the Haskell community) is a classic technique in functional programming that represents an element m of a monoid M (in this paper, written additively) by the function $m + - : M \to M$ it induces through addition. Cayley-transformation helps us because the monoid $M \to M$ has very cheap zero and plus operations: id and $(\circ)$. Afterwards, using a better (sparse) representation for the value in which we collect the final gradient, we can ensure that adding a one-hot value to this gradient collector can be done in logarithmic time.

By now, the differentiated program can compute the gradient with a *logarithmic* overhead over the original program. If a logarithmic overhead is not acceptable, the log-factor in the complexity can be removed by using functional mutable arrays (Section 7).

And then we are done, because we have now obtained a code transformation with the right complexity: the differentiated program computes the gradient of the source program at some input with runtime proportional to the runtime of the source program.

Correctness. Correctness of the resulting AD technique follows in two steps: 1. the naive dual-numbers reverse AD algorithm we start with is correct by a logical relations argument detailed by [Nunes and Vákár 2022b]; 2. we transform this into our final algorithm using a combination of (A) standard optimisations that are well-known to be semantics- (hence correctness-)preserving—notably sparse vectors and in-place array updates—and (B) the custom optimisation of linear factoring, which is semantics-preserving because derivatives (backpropagators) are linear functions.

Comparison to other algorithms. We can relate our technique to that of [Krawiec et al. 2022] by noting that we can replace $\mathbf{D}^1_c[\mathbb{R}] = (\mathbb{R}, \mathbb{R} \multimap c)$ with the isomorphic definition $\mathbf{D}^1_c[\mathbb{R}] = (\mathbb{R}, c)$. This turns the linear factoring rule into a *distributive law* $v \cdot x + v \cdot y \rightsquigarrow v \cdot (x + y)$ that is effectively applied at runtime by using an intensional representation of the cotangent expressions of type c. While their development is presented very differently and the equivalence is not at all clear at first sight, we explain the correspondence in Section 8.

This perspective also makes clear the relationship between our technique and that of [Shaikhha et al. 2019]. Where they try to optimise vectorised forward AD to reverse AD at *compile-time* by using a distributive law (which sometimes succeeds for sufficiently simple programs), our technique proposes a clever way of efficiently applying the distributive law in the required places at *run-time*, giving us the power to always achieve the desired reverse AD behaviour.

Finally, we are now in the position to note the similarity to taping-based AD as in [Kmett and contributors 2021; Krawiec et al. 2022]: the incrementing IDs that we attached to backpropagators earlier give a mapping from $\{0, \ldots, n\}$ to our backpropagators. Furthermore, each backpropagator corresponds to either a primitive arithmetic operation performed in the source program, or to an input value; this already means that we have a tape, in a sense, of all performed primitive operations, albeit in the form of a chain of closures. The optimisation using mutable arrays then eliminates also this last difference, because there we actually reify this tape in a large array.

Types: $\sigma, \tau ::= \mathbb{R} \mid () \mid (\sigma, \tau) \mid \sigma \to \tau \mid \mathsf{Int}$

Terms:
$$\begin{aligned} s, t ::=\ & x \mid () \mid (s, t) \mid \mathrm{fst}(t) \mid \mathrm{snd}(t) \mid s\ t \mid \lambda(x : \tau).\ t \mid \textbf{let}\ x : \tau = s\ \textbf{in}\ t \\ \mid\ & r \qquad \text{(literal } \mathbb{R} \text{ values)} \\ \mid\ & op(t_1, \ldots, t_n) \qquad (op \in \mathrm{Op}_n, \text{ primitive operation application } (\mathbb{R}^n \to \mathbb{R})) \end{aligned}$$

Fig. 4. The source language of all variants of this paper's reverse AD transformation. Int, the type of integers, is added as an example of a type that AD does not act upon.

Types:
$$\begin{aligned} \sigma, \tau ::=\ & \mathbb{R} \mid () \mid (\sigma, \tau) \mid \sigma \to \tau \mid \mathsf{Int} \\ \mid\ & \sigma \multimap \tau \qquad \text{(linear functions)} \end{aligned}$$

Terms:
$$\begin{aligned} s, t ::=\ & x \mid () \mid (s, t) \mid \mathrm{fst}(t) \mid \mathrm{snd}(t) \mid s\ t \mid \lambda(x : \tau).\ t \mid \textbf{let}\ x : \tau = s\ \textbf{in}\ t \mid r \mid op(t_1, \ldots, t_n) \\ \mid\ & \underline{\lambda}(z : \tau).\ b \qquad \text{(linear lambda abstraction } (\tau \text{ a type without function arrows))} \end{aligned}$$

Linear function bodies:
$$\begin{aligned} b ::=\ & () \mid (b, b') \mid \mathrm{fst}(b) \mid \mathrm{snd}(b) && \text{(tupling)} \\ \mid\ & z && \text{(reference to } \underline{\lambda}\text{-bound variable)} \\ \mid\ & x\ b && \text{(linear function application; } x : \sigma \multimap \tau \text{ is an identifier)} \\ \mid\ & \partial_i op(x_1, \ldots, x_n)(b) && (op \in \mathrm{Op}_n, \ i\text{'th partial derivative of } op\ (\mathbb{R}^n \to \mathbb{R})) \\ \mid\ & b + b' && \text{(elementwise addition of results)} \\ \mid\ & \underline{0} && \text{(zero of result type)} \end{aligned}$$

Fig. 5. The target language of the unoptimised variant of the reverse AD transformation. Components that are also in the source language (Fig. 4) are set in grey.

3 NAIVE, UNOPTIMISED DUAL-NUMBERS REVERSE AD

We first describe the naive implementation of dual-numbers reverse AD: this algorithm is easy to define and prove correct compositionally, but it is wildly inefficient in terms of complexity. Indeed, it tends to blow up to exponential overhead over the original function, whereas the desired complexity is to have only a constant factor overhead over the original function. Later, we will apply a number of optimisations to this algorithm (in Section 4 and onwards) that fix the complexity issues, to derive an algorithm that does have the desired complexity.

3.1 Source and Target Languages

The reverse AD methods in this paper are code transformations, and hence have a source language (in which input programs may be written) and a target language (in which gradient programs are expressed). While the source language will be identical for all versions of the transformation that we discuss, the target language will expand to support the optimisations that we perform.

The source language is defined in Fig. 4; the initial target language is given in Fig. 5. The typing of the source language is completely standard, so we omit typing rules here. We assume call-by-value evaluation. The only part that warrants explanation is the treatment of primitive operations: for all $n \in \mathbb{Z}_{>0}$ we presume the presence of a set Op_n containing primitive operations op of type $\mathbb{R}^n \to \mathbb{R}$ in the source language. The program transformation does not care what the contents of Op_n are, as long as their derivatives are available in the target language after differentiation.

In the target language in Fig. 5, we add linear functions with the type $\sigma \multimap \tau$: these functions are linear in the sense of being monoid homomorphisms, meaning that if $f : \sigma \multimap \tau$, then $[\![f]\!](0) = 0$ and $[\![f]\!](x + y) = [\![f]\!](x) + [\![f]\!](y)$. Because it is not well-defined what the derivative of a function value (in the input or output of a program) should be, we disallow function types on either side

On types: $\mathbf{D}^1_c[\mathbb{R}] = (\mathbb{R}, \mathbb{R} \multimap c) \qquad \mathbf{D}^1_c[()] = () \qquad \mathbf{D}^1_c[(\sigma, \tau)] = (\mathbf{D}^1_c[\sigma], \mathbf{D}^1_c[\tau])$

$$\mathbf{D}^1_c[\sigma \to \tau] = \mathbf{D}^1_c[\sigma] \to \mathbf{D}^1_c[\tau] \qquad \mathbf{D}^1_c[\mathsf{Int}] = \mathsf{Int}$$

On environments: $\mathbf{D}^1_c[\varepsilon] = \varepsilon \qquad \mathbf{D}^1_c[\Gamma, x : \tau] = \mathbf{D}^1_c[\Gamma], x : \mathbf{D}^1_c[\tau]$

On terms:

If $\Gamma \vdash t : \tau$ then $\mathbf{D}^1_c[\Gamma] \vdash \mathbf{D}^1_c[t] : \mathbf{D}^1_c[\tau]$

$\mathbf{D}^1_c[x : \tau] = x : \mathbf{D}^1_c[\tau]$ $\qquad$ $\mathbf{D}^1_c[()] = ()$

$\mathbf{D}^1_c[(s, t)] = (\mathbf{D}^1_c[s], \mathbf{D}^1_c[t])$ $\qquad$ $\mathbf{D}^1_c[\mathrm{fst}(t)] = \mathrm{fst}(\mathbf{D}^1_c[t])$

$\mathbf{D}^1_c[\mathrm{snd}(t)] = \mathrm{snd}(\mathbf{D}^1_c[t])$ $\qquad$ $\mathbf{D}^1_c[s\ t] = \mathbf{D}^1_c[s]\ \mathbf{D}^1_c[t]$

$\mathbf{D}^1_c[\lambda(x : \tau).\ t] = \lambda(x : \mathbf{D}^1_c[\tau]).\ \mathbf{D}^1_c[t]$ $\qquad$ $\mathbf{D}^1_c[\mathbf{let}\ x : \tau = s\ \mathbf{in}\ t] = \mathbf{let}\ x : \mathbf{D}^1_c[\tau] = \mathbf{D}^1_c[s]\ \mathbf{in}\ \mathbf{D}^1_c[t]$

$\mathbf{D}^1_c[r] = (r, \underline{\lambda}(z : \mathbb{R}).\ \underline{0})$

$$\mathbf{D}^1_c[op(t_1, \ldots, t_n)] = \mathbf{let}\ (x_1, d_1) = \mathbf{D}^1_c[t_1]\ \mathbf{in}\ \ldots\ \mathbf{in}\ \mathbf{let}\ (x_n, d_n) = \mathbf{D}^1_c[t_n]$$
$$\mathbf{in}\ (op(x_1, \ldots, x_n)$$
$$, \underline{\lambda}(z : \mathbb{R}).\ d_1\ (\partial_1 op(x_1, \ldots, x_n)(z)) + \cdots + d_n\ (\partial_n op(x_1, \ldots, x_n)(z)))$$

Fig. 6. The naive code transformation from the source (Fig. 4) to the target (Fig. 5) language. The cases where $\mathbf{D}^1_c$ just maps homomorphically over the source language are set in gray.

of the $\multimap$-arrow.[1] (Note that higher-order functions *within* the program are fine; the full program should just have first-order input and output types.) Operationally, however, linear functions are just regular functions: the operational meaning of all code in this paper remains identical if all $\multimap$-arrows are replaced with $\to$ (and partial derivative operations are allowed in regular terms).

On the term level, we add an introduction form for linear functions; because we disallowed linear function types from or to function spaces, neither τ nor the type of b can contain function types in $\underline{\lambda}(z : \tau).\ b$. The body of such linear functions is given by the restricted term language under b, which adds application of linear functions (identified by a variable reference), partial derivative operators, and zero and plus operations, but removes variable binding and lambda abstraction.

Note that zero and plus will always be of a type that is (part of) the domain or codomain of a linear function, which therefore has the required commutative monoid structure. The fact that these two operations are not constant-time will be addressed when we improve the complexity of our algorithm later.

Regarding the derivatives of primitive operations: in a linear function, we must be able to compute the linear (reverse) derivatives of the primitive operations. For every $op \in \mathrm{Op}_n$ we require the availability of $\partial_i op : \mathbb{R}^n \to (\mathbb{R} \multimap \mathbb{R})$ with the semantics $[\![\partial_i op]\!](x)(d) = d \cdot \frac{\partial([\![op]\!](x))}{\partial x_i}$.

3.2 The Code Transformation

The naive dual-numbers reverse AD algorithm acts homomorphically over all program constructs in the input program, except for those constructs that non-trivially manipulate real scalars. The full program transformation is given in Fig. 6. Here, we use some syntactic sugar: $\mathbf{let}\ (x_1, x_2) = s\ \mathbf{in}\ t$ should be read as $\mathbf{let}\ y = s\ \mathbf{in}\ \mathbf{let}\ x_1 = \mathrm{fst}(y)\ \mathbf{in}\ \mathbf{let}\ x_2 = \mathrm{snd}(y)\ \mathbf{in}\ t$, where y is fresh.

The transformation consists of a mapping $\mathbf{D}^1_c[\tau]$ on types τ and a mapping $\mathbf{D}^1_c[t]$ on terms t.[2] The mapping on types works homomorphically except on scalars, which it maps (in the style of dual-numbers AD) to a *pair* of a scalar and a derivative of that scalar. In constrast to forward AD,

[1] In Section 5 we will, actually, put endomorphisms $(a \to a)$ on both sides of a $\multimap$-arrow; for justification, see there.

[2] We will choose c to be the domain type of the top-level program; later we will modify c to support our optimisations.

however, the derivative is not represented by another scalar (which in forward AD would contain the derivative of this scalar result with respect to a particular initial input value), but instead by a *backpropagator* that maps the reverse derivative of this scalar (the partial derivative of the final result with respect to this scalar) to the reverse derivative of the full input, assuming that the result depends only on the input through this scalar. (See Section 3.3.)

Variable references, tuples, projections, function application, lambda abstraction and let-binding are mapped homomorphically, i.e., the code transformation simply recurses over the subterms of the current term. However, note that for variable references, lambda abstraction and let-binding, the types of the variables do change.

Scalar constants are transformed to a pair of that scalar constant and a backpropagator that produces the derivative of the input given the derivative of this subterm. This input derivative is zero, since the current subterm (a scalar constant) does not depend on the input.

Finally, primitive scalar operations are the most important place where this code transformation does something non-trivial. First, we compute the values and backpropagators of the (scalar) arguments to the operation, after which we can compute the original (scalar) result by applying the original operation to those argument values. Second, we return the backpropagator of the result of the operation, which applies all partial derivatives of the primitive operation to the incoming cotangent derivative, passes the results on to the corresponding backpropagators of the arguments, and finally adds the (top-level input derivative) results of type c together.

3.3 Wrapper of the AD Transformation: Exposing a Usable API

The ideal API of reverse AD. Given a program $(\lambda(x : \sigma).\ t) : \sigma \to \tau$ that computes a differentiable function $[\![\lambda(x : \sigma).\ t]\!] : [\![\sigma]\!] \to [\![\tau]\!]$, where σ, τ do not contain function types, we expect the following type for its its reverse derivative:

$$\mathrm{Wrap}^1[\lambda(x : \sigma).\ t] : \sigma \to (\tau, \underline{\tau} \multimap \underline{\sigma})$$

Here, we write $\underline{\tau}$ and $\underline{\sigma}$ for the types of cotangent vectors to τ and σ; for example, $\underline{\mathbb{R}} = \mathbb{R}$ and $\underline{(\sigma, \tau)} = (\underline{\sigma}, \underline{\tau})$. The idea is that $\mathrm{Wrap}^1[\lambda(x : \sigma).\ t]$ computes the function $x \mapsto (\overline{[\![\lambda(x : \sigma).\ t]\!](x)}, v \mapsto \overline{D_x[\![\lambda(x : \sigma).\ t]\!]^t(v)})$, where we write $D_x f^t$ for the transposed derivative of f at the point x.

A step towards the right type. This type seems quite different from that of our AD transformation of Fig. 6. Suppose that our AD transformation instead had the following type:

$$\widetilde{\mathbf{D}}^1_c[\lambda(x : \sigma).\ t] : (\sigma, \underline{\sigma} \multimap c) \to (\tau, \underline{\tau} \multimap c),$$

Then by choosing $c = \sigma$ and passing $\mathrm{id} : \underline{\sigma} \multimap \underline{\sigma}$, we can construct our desired wrapper with the definition $\mathrm{Wrap}^1[\lambda(x : \sigma).\ t] = \lambda(y : \sigma).\ \widetilde{\mathbf{D}}^1_\sigma[\lambda(x : \sigma).\ t]\ (y, \mathrm{id})$.

(De)interleaving backpropagators. The typing of $\widetilde{\mathbf{D}}^1_c[\lambda(x : \sigma).\ t]$ feels similar to the typing of the transform in Fig. 6:

$$\mathbf{D}^1_c[\lambda(x : \sigma).\ t] : \mathbf{D}^1_c[\sigma] \to \mathbf{D}^1_c[\tau]$$

Indeed, the missing part is interleaving and deinterleaving of the backpropagators, implementing the isomorphism of linear function types $(\underline{\sigma}, \underline{\tau}) \multimap c \cong (\underline{\sigma} \multimap c, \underline{\tau} \multimap c)$. By doing such interleaving, we can define:

$$\widetilde{\mathbf{D}}^1_c[\lambda(x : \sigma).\ t] = \lambda(y : (\sigma, \underline{\sigma} \multimap c)).\ \mathrm{Deinterleave}^1_\tau\ (\mathbf{D}^1_c[\lambda(x : \sigma).\ t]\ (\mathrm{Interleave}^1_\sigma\ y))$$

Suitable definitions of $\mathrm{Interleave}^1$ and $\mathrm{Deinterleave}^1$ are shown in Fig. 7, including the resulting definition of Wrap^1 (where we already inlined the transformation for the top-level lambda).

$$
\begin{aligned}
&\mathrm{Interleave}^1_{\tau} &&: \forall c.\ (\tau, \tau \multimap c) \to \mathbf{D}^1_c[\tau] \\
&\mathrm{Interleave}^1_{\mathbb{R}} &&= \lambda(x, d).\ (x, d) \\
&\mathrm{Interleave}^1_{()} &&= \lambda((), d).\ () \\
&\mathrm{Interleave}^1_{(\sigma,\tau)} &&= \lambda((x, y), d).\ (\mathrm{Interleave}^1_{\sigma}\ (x, \underline{\lambda}(z : \sigma).\ d\ (z, \underline{0})) \\
&&&\qquad ,\mathrm{Interleave}^1_{\tau}\ (y, \underline{\lambda}(z : \tau).\ d\ (\underline{0}, z))) \\
&\mathrm{Interleave}^1_{\mathsf{Int}} &&= \lambda(n, d).\ n \\
&\mathrm{Interleave}^1_{\sigma\to\tau} &&= \text{not defined!}
\end{aligned}
$$

$$
\begin{aligned}
&\mathrm{Deinterleave}^1_{\tau} &&: \forall c.\ \mathbf{D}^1_c[\tau] \to (\tau, \tau \multimap c) \\
&\mathrm{Deinterleave}^1_{\mathbb{R}} &&= \lambda(x, d).\ (x, d) \\
&\mathrm{Deinterleave}^1_{()} &&= \lambda().\ ((), \underline{\lambda}(z : ()).\ \underline{0}) \\
&\mathrm{Deinterleave}^1_{(\sigma,\tau)} &&= \lambda(x, y).\ \mathbf{let}\ (x_1, x_2) = \mathrm{Deinterleave}^1_{\sigma}\ x \\
&&&\qquad \mathbf{in\ let}\ (y_1, y_2) = \mathrm{Deinterleave}^1_{\tau}\ y \\
&&&\qquad \mathbf{in}\ ((x_1, y_1), \underline{\lambda}(z : (\sigma, \tau)).\ x_2\ (\mathrm{fst}(z)) + y_2\ (\mathrm{snd}(z))) \\
&\mathrm{Deinterleave}^1_{\mathsf{Int}} &&= \lambda n.\ (n, \underline{\lambda}(z : \mathsf{Int}).\ \underline{0}) \\
&\mathrm{Deinterleave}^1_{\sigma\to\tau} &&= \text{not defined!}
\end{aligned}
$$

$$
\begin{aligned}
&\mathrm{Wrap}^1 : (\sigma \to \tau) \rightsquigarrow (\sigma \to (\tau, \tau \multimap \sigma)) \\
&\mathrm{Wrap}^1[\lambda(x : \sigma).\ t] = \lambda(x : \sigma).\ \mathbf{let}\ x : \mathbf{D}^1_{\sigma}[\sigma] = \mathrm{Interleave}^1_{\sigma}\ (x, \mathrm{id})\ \mathbf{in}\ \mathrm{Deinterleave}^1_{\tau}\ (\mathbf{D}^1_{\sigma}[t])
\end{aligned}
$$

Fig. 7. Wrapper around $\mathbf{D}^1_c$ of Fig. 6.

3.4 Complexity of the Naive Transformation

Reverse AD transformations like the one described in this section are well-known to be correct (e.g. [Brunel et al. 2020; Huot et al. 2020; Mazza and Pagani 2021; Nunes and Vákár 2022b]). However, as given here, it does not at all have the right time complexity.

The forward pass is fine: computing the value of $\mathrm{Wrap}^1[\lambda(x : \sigma).\ t : \tau] : \sigma \to (\tau, \tau \multimap \sigma)$ takes time proportional to the original program t. However, the problem arises when we call the top-level backpropagator returned by the wrapper. When we do so, we start a tree of calls to the linear backpropagators of all scalars in the program, where the backpropagator corresponding to a particular scalar value will be invoked once for each usage of that scalar as an argument to a primitive operation. This means that any sharing of scalars in the original program results in multiple calls to the same backpropagator in the derivative program. Fig. 2 displays an example program t with its naive derivative $\mathbf{D}^1_c[t]$, in which sharing of scalars results in exponential time complexity.

This overhead is unacceptable: we can do much better. For first-order programs, we understand well how to write a code transformation such that the output program computes the gradient in only a constant factor overhead over the original program [Griewank and Walther 2008]. This is less immediately clear for higher-order programs, as we consider here, but it is nevertheless possible.

In [Brunel et al. 2020], this problem of exponential complexity is addressed by observing that calling a linear backpropagator multiple times is actually a waste of work: indeed, linearity of a backpropagator f means that $f\ x + f\ y = f\ (x + y)$. Hopefully, applying this *linear factoring rule* from left to right (thereby taking together two calls into one) allows us to ensure that every backpropagator is executed at most once.

And indeed, should we achieve this, the complexity issue described above (the exponential blowup) is fixed: every created backpropagator corresponds to some computation in the original program (either a primitive operation, a scalar constant or an input value), so with maximal

application of linear factoring, the number of backpropagator executions would become proportional to the runtime of the original program. If we can further make the body of a single backpropagator (not counting its callees) constant-time,[3] the differentiated program will compute the gradient with only a constant-factor overhead over the original program—as it should be for reverse AD.

However, this argument crucially depends on us being able to ensure that every backpropagator gets invoked at most once. The solution of [Brunel et al. 2020] is to symbolically evaluate the output program of the transformation to a straight-line program with the input backpropagators still as symbolic variables, and afterwards symbolically reduce the obtained straight-line program in a very specific way, making use of the linear factoring rule ($f\ x + f\ y = f\ (x + y)$) in judicious places.

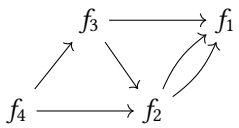
In this paper, we present an alternative way to use linear factoring to make standard, call-by-value evaluation of the target language have the correct computational complexity without any need for symbolic execution. We achieve this by changing the type c that the input backpropagators map to, to a more intelligent type than the space of cotangents of the input that we have considered so far.

4 LINEAR FACTORING BY STAGING FUNCTION CALLS

As observed above in Section 3.4, the most important complexity problem of the reverse AD algorithm is solved if we ensure that all backpropagators are invoked at most once, and for that we must use that every linear function f satisfies $f\ x + f\ y = f\ (x + y)$. This means that we must find a way to "merge" all invocations of a single backpropagator using this linear factoring rule so that in the end only one invocation remains (or zero if it was not invoked at all in the first place).

Evaluation order. Ensuring this complete merging of linear function calls is really a question of choosing an order of evaluation for the tree of function calls created by the backpropagators. Consider for example the (typical) situation where a program generates the following backpropagators:

$$
\begin{aligned}
f_1 &= \lambda(z : \mathbb{R}).\ (0, (z, 0)) \\
f_2 &= \lambda(z : \mathbb{R}).\ f_1\ (2 \cdot z) + f_1\ (3 \cdot z) \\
f_3 &= \lambda(z : \mathbb{R}).\ f_2\ (4 \cdot z) + f_1\ (5 \cdot z) \\
f_4 &= \lambda(z : \mathbb{R}).\ f_2\ z + f_3\ (2 \cdot z)
\end{aligned}
$$



and where f_4 is the (only) backpropagator contained in the result. Normal call-by-value evaluation of f_4 would yield two invocations of f_2 and five invocations of f_1, following the displayed call graph.

However, taking inspiration from symbolic evaluation and moving away from standard call-by-value for a moment, we could also first invoke f_3 to expand the body of f_4 to $f_2\ z + f_2\ (4 \cdot (2 \cdot z)) + f_1\ (5 \cdot (2 \cdot z))$. Now we can take the two invocations of f_2 together using linear factoring to produce $f_2\ (z + 4 \cdot (2 \cdot z)) + f_1\ (5 \cdot (2 \cdot z))$; then invoking f_2 first, producing two more calls to f_1, we are left with three calls to f_1 which we can take together to a single call using linear factoring, which we can then evaluate. With this alternate evaluation order, we have indeed ensured that every linear function is invoked at most (in this case, exactly) once.

If we want to obtain something like this evaluation order, the first thing that we must achieve is to *postpone* invocation of linear functions until we conclude that we have merged all calls to that function and that its time for evaluation has arrived. To achieve this goal, we would like to change the representation of c to a dictionary mapping linear functions to the argument at which we intend to later call them.[4] Note that this uniform representation in a dictionary works because all backpropagators have the same codomain. The idea is that we replace what are now applications of linear functions with creation of a dictionary containing one key-value (function-argument) pair,

[3] Obstacles to this are e.g. $\underline{0}$ and $(+)$ on the type c; we will fix this in Sections 5 to 7.
[4] This is the intuition; it will not go through precisely as planned, but something similar will.

and to replace addition of values in c with taking the union of dictionaries, where arguments for common keys are added together.

Initial Staged object. More concretely, we want to replace the c in $\mathbf{D}_c^1[\mathbb{R}] = (\mathbb{R}, \mathbb{R} \multimap c)$ with 'Staged c' (our "dictionary"), which we define as follows: ('Map' being the usual persistent tree-map)

$$\text{Staged}\, c = (c, \text{Map}\ (\mathbb{R} \multimap \text{Staged}\, c)\ \mathbb{R})$$

This type can represent both literal c values (necessary for the one-hot vectors returned by the input backpropagators created in Interleave^1) and staged (delayed) calls to linear functions. We use Map to denote a standard (persistent) tree-map as found in every functional language. The intuitive semantics of a value $(x, \{f_1 \mapsto a_1, f_2 \mapsto a_2\}) : \text{Staged}\, c$ is its *resolution* $x + f_1\ a_1 + f_2\ a_2 : c$.

To be able to replace c with Staged c in $\mathbf{D}_c^1$, we must support all operations that we perform on c also on Staged c. We implement them as follows:

- $\underline{0} : c$ becomes simply $0_{\text{Staged}} \coloneqq (\underline{0}, \{\}) : \text{Staged}\, c$.
- $(+) : c \to c \to c$ becomes $(+_{\text{Staged}})$, adding c values using $(+)$ and taking the union of the two Maps. **Here we apply linear factoring**: if the two Maps both have a value for the same key (i.e. we have two staged invocations to the same linear function f), the resulting map will have *one* value for that same key f: the sum of the arguments stored in the two separate Maps. For example:

$$(c_1, \{f_1 \mapsto a_1, f_2 \mapsto a_2\}) +_{\text{Staged}} (c_2, \{f_2 \mapsto a_3\}) = (c_1 + c_2, \{f_1 \mapsto a_1, f_2 \mapsto a_2 + a_3\})$$

- The one-hot c values created in the backpropagators from Interleave^1 are stored in the c component of Staged c.
- An application $f\ x$ of a backpropagator $f : \mathbb{R} \multimap c$ to an argument $x : \mathbb{R}$ now gets replaced with $\text{SCall}\ f\ x \coloneqq (\underline{0}, \{f \mapsto x\}) : \text{Staged}\, c$. This occurs in $\mathbf{D}_c^1[op(\ldots)]$ and in Deinterleave^1.

What is missing from this list is how to "resolve" the final Staged c value produced by the derivative computation down to a plain c value—we need this at the end of the wrapper. This resolve algorithm will need to call functions in the Staged c object in the correct order, ensuring that we only invoke a backpropagator when we are sure that we have collected all calls to it in the Map. For example, in the example at the beginning of this section, $f_4\ 1$ returns $(\underline{0}, \{f_2 \mapsto 1, f_3 \mapsto 2\})$. At this point, "resolving f_3" means calling f_3 at 2, observing the return value $(\underline{0}, \{f_2 \mapsto 8, f_1 \mapsto 10\})$, and adding it to the remainder (i.e. without the f_3 entry) of the previous Staged c object to get $(\underline{0}, \{f_2 \mapsto 9, f_1 \mapsto 10\})$.

But as we observed above, the choice of which function to invoke first is vital to the complexity of the reverse AD algorithm: if we chose f_2 first instead of f_3, the later call to f_3 would produce another call to f_2, forcing us to evaluate f_2 twice—something that we must avoid. There is currently no information in a Staged c object from which we can deduce the correct order of invocation, so we need something extra.

There is another problem with the current definition of Staged c: it contains a Map keyed by functions, meaning that we need equality—actually, even an ordering—on functions! This is nonsense in general. Fortunately, both problems can be tackled with the same solution.

Resolve order. The backpropagators that occur in the derivative program (as produced by $\mathbf{D}_c^1$ from Fig. 6) are not just arbitrary functions. Indeed, taking the target type c of the input backpropagators to be equal to the input type σ of the original program (of type $\sigma \to \tau$), as we do in Wrap^1 in Fig. 7, all backpropagators in the derivative program have one of the following three forms:

(1) $(\underline{\lambda}(z : \mathbb{R}).\ t)$ where t is a tuple (of type σ) filled with zero scalars except for one position, where it places z; we call such tuples *one-hot tuples*. These backpropagators result from $\text{Interleave}_\sigma^1$ (Fig. 7) after trivial beta-reduction of the intermediate linear functions.

(2) $(\underline{\lambda}(z : \mathbb{R}).\ \underline{0})$ occurs as the backpropagator of a scalar constant r. Note that since this $\underline{0}$ is of type σ, operationally it is equivalent to a tuple filled completely with zero scalars.
(3) $(\underline{\lambda}(z : \mathbb{R}).\ d_1\ (\partial_1 op(x_1, \ldots, x_n)(z)) + \cdots + d_n\ (\partial_n op(x_1, \ldots, x_n)(z)))$ for an $op \in \mathrm{Op}_n$ where $d_1, \ldots, d_n$ are other linear backpropagators: these occur as the backpropagators generated for primitive operations.

Insight: Hence, we observe that all other backpropagators that a backpropagator f will ever call *are contained in its closure*, and were hence created (at runtime of the derivative program) before f itself was created. Therefore, if we give all backpropagators at runtime sequentially incrementing integer IDs, where a $\underline{\lambda}$ allocated later in time gets a higher ID, we obtain that a called backpropagator always has a *lower* ID than the backpropagator it was called from.

This provides an answer to the question of in what order to resolve backpropagators: if they have sequentially increasing IDs, just start with the one with the largest ID! After all, any calls to other backpropagators that it produces in the returned Staged c value will have lower IDs, and so cannot be functions that we have already resolved (i.e. called) before. And as promised, giving backpropagators IDs also solves the issue of using functions as keys in a Map: we can just use the (integer) ID as the Map key, which is perfectly valid and efficient.

With this knowledge, we rewrite Staged c and SCall to the following:

$$\begin{aligned}
&\mathrm{Staged}\, c = (c, \mathrm{Map}\ \mathsf{Int}\ (\mathbb{R} \multimap \mathrm{Staged}\, c, \mathbb{R})) \\
&\mathrm{SCall} \quad : (\mathsf{Int}, \mathbb{R} \multimap \mathrm{Staged}\, c) \to \mathbb{R} \multimap \mathrm{Staged}\, c \\
&\mathrm{SCall}\ (i, f)\ x = (\underline{0}, \{i \mapsto (f, x)\})
\end{aligned}$$

We call the second component of a Staged c value, which has type Map Int $(\mathbb{R} \multimap \mathrm{Staged}\, c, \mathbb{R}))$, the *staging map*, after its function to stage (linear) function calls.

The only thing that remains is to actually generate the IDs for the backpropagators at runtime; this we do using an ID generation monad (a state monad with a state of type Int). The resulting new program transformation, modified from Figs. 6 and 7, is shown in Fig. 8.

New program transformation. In Fig. 8, the term transformation now produces a term in the ID generation monad $(\mathsf{Int} \to (-, \mathsf{Int}))$; therefore, all functions in the original program will also need to run in the same monad. This gives the second change in the type transformation (aside from $\mathbf{D}^2_c[\mathbb{R}]$, which now tags backpropagators with an ID): $\mathbf{D}^2_c[\sigma \to \tau]$ now produces a monadic function type instead of a plain function type.

On the term level, notice that the backpropagator for primitive operations (in $\mathbf{D}^2_c[op(...)]$) now no longer calls $d_1, \ldots, d_n$ (the backpropagators of the arguments to the operation) directly, but instead registers the calls as pairs of function and argument in the Staged c returned by the backpropagator. The $\cup$ in the definition of $(+_{\mathrm{Staged}})$ refers to map union including linear factoring; for example:

$$\{i_1 \mapsto (f_1, a_1), i_2 \mapsto (f_2, a_2)\} \cup \{i_2 \mapsto (f_2, a_3)\} = \{i_1 \mapsto (f_1, a_1), i_2 \mapsto (f_2, a_2 + a_3)\}$$

Note that the transformation assigns consistent IDs to backpropagators: it will never occur that two staging maps have an entry with the same key (ID) but with a different function in the value.

In the wrapper, Interleave^2 is lifted into the monad and generates IDs for scalar backpropagators; Deinterleave^2 is essentially unchanged. The initial backpropagator provided to Interleave^2 in Wrap^2, which was $\mathrm{id} : \sigma \multimap \sigma$ in Fig. 7, has now become $\mathrm{SCotan} : \sigma \multimap \mathrm{Staged}\, \sigma$, which injects a cotangent into a Staged c object. Interleave^2 will "split" this function up into individual $\mathbb{R} \multimap \mathrm{Staged}\, \sigma$ backpropagators for each of the individual scalars in σ.

At the end of the wrapper, we apply the insight that we had earlier: by resolving (calling and eliminating) the backpropagators in the final Staged c returned by the differentiated program in

On types:

$$\mathbf{D}_c^2[\mathbb{R}] = (\mathbb{R}, (\mathsf{Int}, \mathbb{R} \multimap \mathsf{Staged}\, c)) \qquad \mathbf{D}_c^2[()] = () \qquad \mathbf{D}_c^2[(\sigma,\tau)] = (\mathbf{D}_c^2[\sigma], \mathbf{D}_c^2[\tau])$$
$$\mathbf{D}_c^2[\sigma \to \tau] = \mathbf{D}_c^2[\sigma] \to \mathsf{Int} \to (\mathbf{D}_c^2[\tau], \mathsf{Int}) \qquad \mathbf{D}_c^2[\mathsf{Int}] = \mathsf{Int}$$

On terms:

If $\Gamma \vdash t : \tau$ then $\mathbf{D}_c^2[\Gamma] \vdash \mathbf{D}_c^2[t] : \mathsf{Int} \to (\mathbf{D}_c^2[\tau], \mathsf{Int})$

$\mathbf{D}_c^2[x : \tau] = \lambda i.\ (x : \mathbf{D}_c^2[\tau], i)$

$\mathbf{D}_c^2[(s,t)] = \lambda i.\ \mathbf{let}\ (x, i') = \mathbf{D}_c^2[s]\ i\ \mathbf{in}\ \mathbf{let}\ (y, i'') = \mathbf{D}_c^2[t]\ i'\ \mathbf{in}\ ((x,y), i'')$

$\mathbf{D}_c^2[\mathbf{let}\ x : \tau = s\ \mathbf{in}\ t] = \lambda i.\ \mathbf{let}\ (x : \mathbf{D}_c^2[\tau], i') = \mathbf{D}_c^2[s]\ i\ \mathbf{in}\ \mathbf{D}_c^2[t]\ i'$

etc.

$\mathbf{D}_c^2[r] = \lambda i.\ ((r, (i, \underline{\lambda}(z : \mathbb{R}).\ 0_{\mathsf{Staged}})), i + 1)$

$\mathbf{D}_c^2[op(t_1, \ldots, t_n)] =$

$\quad \lambda i.\ \mathbf{let}\ ((x_1, d_1), i_1) = \mathbf{D}_c^2[t_1]\ i\ \mathbf{in}\ \ldots\ \mathbf{in}\ \mathbf{let}\ ((x_n, d_n), i_n) = \mathbf{D}_c^2[t_n]\ i_{n-1}$

$\quad\quad \mathbf{in}\ ((op(x_1, \ldots, x_n), (i_n, \underline{\lambda}(z : \mathbb{R}).\ \mathsf{SCall}\ d_1\ (\partial_1 op(x_1, \ldots, x_n)(z)) +_{\mathsf{Staged}} \cdots +_{\mathsf{Staged}}$

$\quad\quad\quad\quad \mathsf{SCall}\ d_n\ (\partial_n op(x_1, \ldots, x_n)(z)))$

$\quad\quad , i_n + 1)$

Changed wrapper:

$\mathrm{Wrap}^2 : (\sigma \to \tau) \rightsquigarrow (\sigma \to (\tau, \tau \multimap \sigma))$

$\mathrm{Wrap}^2[\lambda(x : \sigma).\ t] = \lambda(x : \sigma).\ \mathbf{let}\ (x : \mathbf{D}_\sigma^2[\sigma], i) = \mathrm{Interleave}_\sigma^2\ (x, \mathsf{SCotan})\ 0$
$\quad\quad \mathbf{in}\ \mathbf{let}\ (y, d) = \mathrm{Deinterleave}_\tau^2\ (\mathrm{fst}(\mathbf{D}_\sigma^2[t]\ i))$
$\quad\quad \mathbf{in}\ (y, \underline{\lambda}(z : \tau).\ \mathsf{SResolve}\ (d\ z)) \qquad$ *— see main text for SResolve*

$\mathrm{Interleave}_\tau^2 : \forall c.\ (\tau, \tau \multimap \mathsf{Staged}\, c) \to \mathsf{Int} \to (\mathbf{D}_c^2[\tau], \mathsf{Int})$
$\mathrm{Interleave}_{\mathbb{R}}^2 = \lambda(x, d).\ \lambda i.\ ((x, (i, d)), i + 1)$
$\mathrm{Interleave}_{()}^2 = \lambda((), d).\ \lambda i.\ ((), i)$
$\mathrm{Interleave}_{(\sigma,\tau)}^2 = \lambda((x,y), d).\ \lambda i.\ \mathbf{let}\ (x', i') = \mathrm{Interleave}_\sigma^2\ (x, \underline{\lambda}(z : \sigma).\ d\ (z, \underline{0}))\ i$
$\quad\quad \mathbf{in}\ \mathbf{let}\ (y', i'') = \mathrm{Interleave}_\tau^2\ (y, \underline{\lambda}(z : \tau).\ d\ (\underline{0}, z))\ i'$
$\quad\quad \mathbf{in}\ ((x', y'), i'')$
$\mathrm{Interleave}_{\mathsf{Int}}^2 = \lambda(n, d).\ \lambda i.\ (n, i)$

$\mathrm{Deinterleave}_\tau^2$ gets type $\forall c.\ \mathbf{D}_c^2[\tau] \to (\tau, \tau \multimap \mathsf{Staged}\, c)$ and ignores the new Int in $\mathbf{D}_c^2[\mathbb{R}]$. $\underline{0}$ changes to 0_{Staged} and $(+)$ changes to $(+_{\mathsf{Staged}})$.

Staged interface:

$\mathsf{Staged}\, c = (c, \mathsf{Map}\ \mathsf{Int}\ (\mathbb{R} \multimap \mathsf{Staged}\, c, \mathbb{R}))$

$0_{\mathsf{Staged}} : \mathsf{Staged}\, c \qquad (+_{\mathsf{Staged}}) : \mathsf{Staged}\, c \to \mathsf{Staged}\, c \to \mathsf{Staged}\, c$
$0_{\mathsf{Staged}} = (\underline{0}, \{\}) \qquad (c, m) +_{\mathsf{Staged}} (c', m') = (c + c', m \cup m') \qquad$ *— with linear factoring*

$\mathsf{SCotan} : c \multimap \mathsf{Staged}\, c \qquad \mathsf{SCall} : (\mathsf{Int}, \mathbb{R} \multimap \mathsf{Staged}\, c) \to \mathbb{R} \multimap \mathsf{Staged}\, c$
$\mathsf{SCotan}\ c = (c, \{\}) \qquad \mathsf{SCall}\ (i, f)\ x = (\underline{0}, \{i \mapsto (f, x)\})$

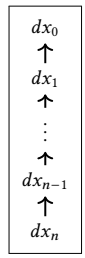
Fig. 8. The monadically transformed code transformation (from Fig. 4 to Fig. 5 plus Staged operations), based on Fig. 6. Parts of $\mathbf{D}_c^2$ and $\mathrm{Interleave}^2$ that were simply lifted to monadic code are set in grey.

order from the highest ID to the lowest ID, we ensure that every backpropagator is called at most once. This is done in the following function:

$$\begin{aligned}&\text{SResolve } (c : \sigma, m : \text{Map Int } (\mathbb{R} \multimap \text{Staged } \sigma, \mathbb{R})) \coloneqq \\ &\quad \textbf{if } m \text{ is empty } \textbf{then } c \\ &\qquad \textbf{else let } i = \text{highest key in } m \\ &\qquad\quad \textbf{in let } (f, a) = \text{lookup } i \text{ in } m \\ &\qquad\quad \textbf{in let } m' = \text{delete } i \text{ from } m \\ &\qquad\quad \textbf{in } \text{SResolve } (f\ a +_{\text{Staged}} (c, m'))\end{aligned}$$

The three operations on m are standard logarithmic-complexity tree-map operations.

4.1 Remaining Complexity Challenges



We have gained a lot with the function call staging so far: where the naive algorithm from Section 3 easily ran into exponential blowup of computation time if the results of primitive operations were used in multiple places, the updated algorithm from Fig. 8 completely solves this issue. For example, the program of Fig. 2 now results in the call graph displayed on the right: each backpropagator is called exactly once. However, some other complexity problems still remain.[5]

In general, for a reverse AD algorithm to have the right complexity, we want the produced derivative program P' to compute the gradient of the original program P at a given input x with runtime only a constant factor times the runtime of P itself on x—and this constant factor should work for all programs P. However, as-is, this requirement is untenable: $P = \mathsf{id} : \tau \to \tau$ always takes constant time whereas its gradient program must at the very least construct the value of P's full (non-zero) gradient, which might be large (indeed, its size is $\text{size}(x)$). Hence, we require that:

$$\begin{aligned}&\exists c > 0.\ \forall P \in \text{Programs}(\sigma \to \tau).\ \forall x : \sigma, dy : \tau. \\ &\qquad \text{cost}(\text{snd}(\text{Wrap}\,[P]\ x)\ dy) \le c \cdot (\text{cost}(P\ x) + \text{size}(x))\end{aligned}$$

where $\text{cost}(E)$ is the time taken to evaluate E to normal form, and $\text{size}(x)$ is the time taken to read all of x sequentially.

So, what is $\text{cost}(\text{snd}(\text{Wrap}\,[P]\ x)\ dy)$? First, the primal pass ($\text{Wrap}\,[P]\ x$) consists of interleaving, running the differentiated program, and deinterleaving.

- Interleave^2 itself runs in $O(\text{size}(x))$. (The backpropagators it creates are more expensive, but those are not called just yet.)
- For the differentiated program, $\mathbf{D}^2_\sigma[P]$, we can see that in all cases of the transformation $\mathbf{D}^2_c$, the right-hand side does the work that P would have done, plus threading of the next ID to generate, as well as creation of backpropagators. Since this additional work is a constant amount per program construct, $\mathbf{D}^2_\sigma[P]$ runs in $O(\text{cost}(P\ x))$.
- Deinterleave^2 runs in $O(\text{size}(P\ x))$, i.e. the size of the program output; this is certainly in $O(\text{cost}(P\ x) + \text{size}(x))$ but likely much less.

Summarising, the primal pass as a whole runs in $O(\text{cost}(P\ x) + \text{size}(x))$, which is as required.

Then, the dual pass ($f\ dy$, where f is the linear function returned by Wrap^2) first calls the backpropagator returned by Deinterleave^2 on the output cotangent, and then passes the result through SResolve to produce the final gradient. Let t be the function body of P (i.e. $P = \lambda(x : \sigma).\ t$).

[5]This section does not provide a proof that Wrap^2 does *not* have the correct complexity; rather, it argues that the expected complexity analysis does not go through. The same complexity analysis *will* go through for Wrap^3 after the improvements of Sections 6 and 7.

- Because the number of scalars in the output is potentially as large as $O(\text{cost}(P\ x) + \text{size}(x))$, the backpropagator returned by Deinterleave^2 is only allowed to perform a constant-time operation for each scalar. However, looking back at Fig. 7, we see that this function calls all scalar backpropagators contained in the result of $\mathbf{D}^2_\sigma[t]$ once, and adds the results using $(+_{\text{Staged}})$. Assuming that the scalar backpropagators run in constant time (we will cover this point later), we are left with the many uses of $(+_{\text{Staged}})$; if these are constant-time, we are still within our complexity budget. However:
  Problem: $(+_{\text{Staged}})$ (see Fig. 8) is not constant-time: it adds values of type c and takes the union of staging maps, both of which may be large.
- Afterwards, we use SResolve on the resulting Staged σ to call every scalar backpropagator in the program (created in $\mathbf{D}^2_\sigma[r]$, $\mathbf{D}^2_\sigma[op(...)]$ and Interleave^2) at most once; this is accomplished using three Map operations and one call to $(+_{\text{Staged}})$ per backpropagator. However, each of the scalar backpropagators corresponds to either a constant-time operation[6] in the original program P or to a scalar in the input x; therefore, in order to stay within the time budget of $O(\text{cost}(P\ x) + \text{size}(x))$, we are only allowed a constant-time overhead per backpropagator here. Since $(+_{\text{Staged}})$ was covered already, we are left with:
  Problem: the Map operations in SResolve are not constant-time.
- While we have arranged to invoke each scalar backpropagator at most once, we still need those backpropagators to individually run in constant-time too: our time budget is $O(\text{cost}(P\ x) + \text{size}(x))$, but there could be $O(\text{cost}(P\ x) + \text{size}(x))$ distinct backpropagators. Recall from earlier that we have three kinds of scalar backpropagators:
  (1) $(\underline{\lambda}(z : \mathbb{R}).\ \text{SCotan}\ (0, \ldots, 0, z, 0, \ldots, 0))$ created in Interleave^2 (with SCotan from Wrap^2).
  Problem: The interleave backpropagators take time $O(\text{size}(x))$, not $O(1)$.
  (2) $(\underline{\lambda}(z : \mathbb{R}).\ 0_{\text{Staged}})$ created in $\mathbf{D}^2_\sigma[r]$.
  Problem: 0_{Staged} takes time $O(\text{size}(x))$, not $O(1)$.
  (3) $(\underline{\lambda}(z : \mathbb{R}).\ \text{SCall}\ d_1\ (\partial_1 op(...)(z)) +_{\text{Staged}} \cdots +_{\text{Staged}} \text{SCall}\ d_n\ (\partial_n op(...)(z)))$ created in $\mathbf{D}^2_\sigma[op(...)]$. Assuming that primitive operation arity is bounded, we are allowed a constant-time operation for each operation argument.
  Problem: SCall creates a $\underline{0} : c$ and therefore runs in $O(\text{size}(x))$, not $O(1)$. (The problem with $(+_{\text{Staged}})$ was already covered above.)

Summarising again, we see that we have three categories of complexity problems to solve:

(A) We are not allowed to perform monoid operations on c as often as we do. (This affects 0_{Staged}, $(+_{\text{Staged}})$ and SCall). Our fix for this (in Section 5) will be to Cayley-transform the Staged c object, including the contained c value, turning zero into id and plus into $(\circ)$ on the type Staged $c \to$ Staged c.

(B) The Interleave backpropagators that create a one-hot c value should avoid touching parts of c that they are zero on. After Cayley-transforming Staged c in Section 5, this problem becomes less pronounced: the backpropagators now *update* a Staged c value, where they can keep untouched subtrees of c fully as-is. However, the one-hot backpropagators will still do work proportional to the *depth* of the program input type c. We will turn this issue into a simple log-factor in the complexity in Section 6 by replacing the c in Staged c with a more efficient structure (namely, Map Int $\mathbb{R}$). This log-factor can optionally be further eliminated using mutable arrays as described in Section 7.

(C) The Map operations in SResolve are logarithmic in the size of the staging map. Like in the previous point, mutable arrays (Section 7) can eliminate this final log-factor in the complexity.

[6] Assuming primitive operations all have bounded arity and are constant-time. A more precise analysis, omitted here, lifts these restrictions—as long as the gradient of a primitive operation can be computed in the same time as the original.

From the analysis above, we can conclude that after we have solved each of these issues, the algorithm attains the correct complexity for reverse AD.

5 CAYLEY-TRANSFORMING THE COTANGENT COLLECTOR

The classical "difference list" trick in functional programming, originally designed to improve the performance of repeated application of the list-append operation [Hughes 1986], is an instance of a more general theorem: any monoid $(M, 0, +)$ is isomorphic to the submonoid of $(M \to M, \mathrm{id}, \circ)$ containing only the functions $\lambda m'.\ m + m'$ for all $m \in M$. In other words, the map that sends $m \in M$ to $(\lambda m'.\ m + m') \in M \to M$ is a monoid homomorphism, and it has a left-inverse: $\lambda f.\ f\ 0$.

In the original application on lists, the intent of moving from $[\tau]$ to $[\tau] \to [\tau]$ (an action that we call *Cayley-transforming* the $[\tau]$ type) was to ensure that the list-append operations are consistently associated to the right. In our case, however, the primary remaining complexity issues are not due to operator associativity, but instead because our monoid has very expensive 0 and + operations (namely, 0_{Staged} and $(+_{\mathrm{Staged}})$). If we Cayley-transform Staged c, i.e. if we replace Staged c with Staged $c \to$ Staged c, all occurrences of 0_{Staged} in the code transformation turn into id and all occurrences of $(+_{\mathrm{Staged}})$ turn into $(\circ)$. Since id is a constant and the composition of two functions can be constructed in constant time, this makes the monoid operations on the codomain of backpropagators (which now becomes Staged $c \to$ Staged c) constant-time.

Hence, all non-trivial work with Staged c objects that we still perform is limited to: 1. the single 0_{Staged} value that the full composition is in the end applied to (to undo the Cayley-transform), and 2. the implementation of the non-monoid operations on Staged c: SCall, SCotan and SResolve. We do not have to worry about one single zero of type c, hence we focus only on SCall, SCotan and SResolve, which get the following updated types after the Cayley-transform:[7] (the changed parts are shown in red)

$$\begin{aligned}
&\mathrm{SCall} &&: (\mathsf{Int}, \mathbb{R} \multimap (\mathrm{Staged}\ c \to \mathrm{Staged}\ c)) \to \mathbb{R} \multimap (\mathrm{Staged}\ c \to \mathrm{Staged}\ c)\\
&\mathrm{SCotan} &&: c \multimap (\mathrm{Staged}\ c \to \mathrm{Staged}\ c)\\
&\mathrm{SResolve} &&: (\mathrm{Staged}\ c \to \mathrm{Staged}\ c) \multimap c
\end{aligned}$$

The definition of Staged c itself also gets changed accordingly:

$$\mathrm{Staged}\ c = (c, \mathrm{Map}\ \mathsf{Int}\ (\mathbb{R} \multimap (\mathrm{Staged}\ c \to \mathrm{Staged}\ c), \mathbb{R}))$$

The new definition of SCall arises from simplifying the composition of the old SCall with $(+_{\mathrm{Staged}})$:

$$\begin{aligned}
\mathrm{SCall}\ (i, f)\ x\ (c, m) = (c, &\text{ if } i \notin m \text{ then insert } i \mapsto (f, x) \text{ into } m\\
&\text{else update } m \text{ at } i \text{ with } (\lambda(_, x').\ (f, x + x')))
\end{aligned}$$

Note that $(+_{\mathrm{Staged}})$ has been eliminated, and we do not use $(+)$ on c here anymore. For SCotan we have to modify the type further (Cayley-transforming its c argument as well) to lose all $(+)$ operations on c:

$$\begin{aligned}
&\mathrm{SCotan} : (c \to c) \multimap (\mathrm{Staged}\ c \to \mathrm{Staged}\ c)\\
&\mathrm{SCotan}\ f\ (c, m) = (f\ c, m)
\end{aligned}$$

SResolve simply applies its argument to 0_{Staged} (undoing the Cayley transform—this is now the only remaining 0_{Staged}) and runs the old code from Section 4, only changing $f\ a +_{\mathrm{Staged}} (c, m')$ to $f\ a\ (c, m')$ on the last line: f from the Map now has type $\mathbb{R} \multimap (\mathrm{Staged}\ c \to \mathrm{Staged}\ c)$.

[7] Despite the fact that we forbade it in Section 3.1, we are putting function types on both sides of a $\multimap$-arrow here. The monoid structure here is the one from the Cayley transform (i.e. with id and $(\circ)$). Notice that this monoid structure is indeed the one we want in this context: the "sum" (composition) of two values of type $(\mathrm{Staged}\ c \to \mathrm{Staged}\ c)$ corresponds with the sum (with $(+_{\mathrm{Staged}})$) of the Staged c values that they represent. (This is the Cayley isomorphism described above.)

On types:

$$\mathbf{D}^3_c[\mathbb{R}] = (\mathbb{R}, (\mathsf{Int}, \mathbb{R} \multimap (\mathsf{Staged}\, c \to \mathsf{Staged}\, c))) \qquad \mathbf{D}^3_c[()] = () \qquad \mathbf{D}^3_c[(\sigma,\tau)] = (\mathbf{D}^3_c[\sigma], \mathbf{D}^3_c[\tau])$$
$$\mathbf{D}^3_c[\sigma \to \tau] = \mathbf{D}^3_c[\sigma] \to \mathsf{Int} \to (\mathbf{D}^3_c[\tau], \mathsf{Int}) \qquad \mathbf{D}^3_c[\mathsf{Int}] = \mathsf{Int}$$

On terms:

$$\text{If } \Gamma \vdash t : \tau \text{ then } \mathbf{D}^3_c[\Gamma] \vdash \mathbf{D}^3_c[t] : \mathsf{Int} \to (\mathbf{D}^3_c[\tau], \mathsf{Int})$$

Same as $\mathbf{D}^2_c$, except with 'id' in place of 0_{Staged} and '$\circ$' in place of $(+_{\mathsf{Staged}})$.

Changed wrapper:

$$\begin{aligned}
&\mathrm{Wrap}^3 : (\sigma \to \tau) \rightsquigarrow (\sigma \to (\tau, \tau \multimap \sigma))\\
&\mathrm{Wrap}^3[\lambda(x:\sigma).\, t] = \lambda(x:\sigma).\ \mathbf{let}\ (x : \mathbf{D}^3_\sigma[\sigma], i) = \mathrm{Interleave}^3_\sigma\ (x, \mathrm{SCotan})\ 0\\
&\qquad\qquad\qquad\qquad \mathbf{in\ let}\ (y, d) = \mathrm{Deinterleave}^3_\tau\ (\mathrm{fst}(\mathbf{D}^3_\sigma[t]\ i))\\
&\qquad\qquad\qquad\qquad \mathbf{in}\ (y, \underline{\lambda}(z:\tau).\ \mathrm{SResolve}\ (d\ z))
\end{aligned}$$

$$\begin{aligned}
&\mathrm{Interleave}^3_\tau : \forall c.\ (\tau, (\tau \to \tau) \multimap (\mathsf{Staged}\, c \to \mathsf{Staged}\, c)) \to \mathsf{Int} \to (\mathbf{D}^3_c[\tau], \mathsf{Int})\\
&\mathrm{Interleave}^3_{\mathbb{R}} = \lambda(x,d).\ \lambda i.\ ((x, (i, \underline{\lambda}(z:\mathbb{R}).\ d\ (\lambda(a:\mathbb{R}).\ z + a))), i+1)\\
&\mathrm{Interleave}^3_{()} = \lambda((), d).\ \lambda i.\ ((), i)\\
&\mathrm{Interleave}^3_{(\sigma,\tau)} = \lambda((x,y),d).\ \lambda i.\\
&\qquad \mathbf{let}\ (x', i') = \mathrm{Interleave}^3_\sigma\ (x, \lambda(f:\sigma\to\sigma).\ d\ (\lambda((v,w):(\sigma,\tau)).\ (f\ v, w)))\ i\\
&\qquad \mathbf{in\ let}\ (y', i'') = \mathrm{Interleave}^3_\tau\ (y, \lambda(f:\tau\to\tau).\ d\ (\lambda((v,w):(\sigma,\tau)).\ (v, f\ w)))\ i'\\
&\qquad \mathbf{in}\ ((x', y'), i'')\\
&\mathrm{Interleave}^3_{\mathsf{Int}} = \lambda(n,d).\ \lambda i.\ (n, i)
\end{aligned}$$

$$\begin{aligned}
&\mathrm{Deinterleave}^3_\tau : \forall c.\ \mathbf{D}^3_c[\tau] \to (\tau, \tau \multimap (\mathsf{Staged}\, c \to \mathsf{Staged}\, c))\\
&(\text{Same as } \mathrm{Deinterleave}^2 \text{ in Fig. 8, except with id and } (\circ) \text{ in place of } 0_{\mathsf{Staged}} \text{ and } (+_{\mathsf{Staged}}))
\end{aligned}$$

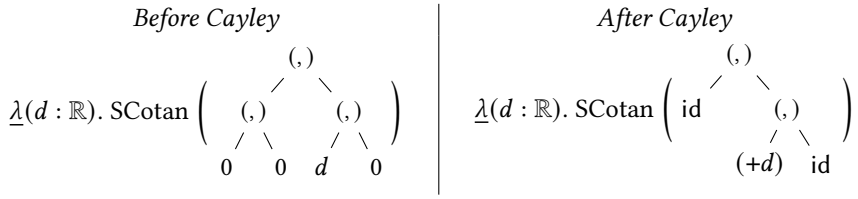
Fig. 9. The Cayley-transformed code transformation, based on Fig. 8. Grey parts are unchanged.

5.1 Code Transformation

The new code transformation is shown in Fig. 9. Aside from the changes to types and to the target monoid of the backpropagators, the only additional change is in Interleave^3, which is adapted to accomodate the additional Cayley-transform to the c argument of SCotan. Note that the backpropagators in Interleave^3 do not create any $\underline{0}$ values for untouched parts of the collected cotangent of type c, as promised, and that the new type of SCotan has indeed eliminated all uses of $(+)$ on c, not just moved them around.

5.2 Remaining Complexity Challenges

In Section 4.1, we pinpointed the three remaining complexity issues with the reverse AD algorithm after function call staging: costly monoid operations on c, costly one-hot backpropagators from Interleave, and logarithmic Map operations in SResolve. The first issue has been solved by Cayley-transforming Staged c: only $\underline{0} : c$ is still used, and that only once (in the new SResolve). For the second issue, although performance of the one-hot backpropagators has improved in most cases, it is still unsatisfactory; for example, given the input type $\sigma = ((\mathbb{R}, \mathsf{Int}), (\mathbb{R}, \mathbb{R}))$, the backpropagator for the second scalar looks as follows before and after the Cayley transform:



This yields complexity logarithmic in the size of the input if the input is balanced, but can degrade to linear in the size of the input in the worst case—which is no better than the previous version. We will make these backpropagators properly logarithmic in the size of the input in Section 6, after which one can remove the final log-factors from the algorithm's complexity by introducing mutable arrays as in Section 7.

6 KEEPING JUST THE SCALARS: EFFICIENT GRADIENT UPDATES

The codomain of the backpropagators is currently $\mathrm{Staged}\, c \to \mathrm{Staged}\, c$, with $\mathrm{Staged}\, c$ defined as:

$$\mathrm{Staged}\, c = (c, \mathsf{Map\ Int}\ (\mathbb{R} \multimap (\mathrm{Staged}\, c \to \mathrm{Staged}\, c), \mathbb{R}))$$

The final cotangent of the input to the program is collected in the first component of the pair, of type c. This collector is initialised with $\underline{0} : c$ in 0_{Staged}, and added to by the one-hot input backpropagators from Interleave, called in SResolve. The task of these input backpropagators is to add the cotangent (of type $\mathbb{R}$) that they receive in their argument, to a particular scalar in the collector.

Hence, all we need of c in $\mathrm{Staged}\, c$ is really the collection of its scalars: the rest simply came from $\underline{0} : c$ and is never changed.[8] Furthermore, the reason why the one-hot input backpropagators currently do not finish in logarithmic time is that c may not be a balanced tree of its scalars. But if we are interested only in the scalars anyway, we can *make* the collector balanced—by replacing it with $\mathsf{Map\ Int}\ \mathbb{R}$:

$$\mathrm{Staged}\, c = (\mathsf{Map\ Int}\ \mathbb{R}, \mathsf{Map\ Int}\ (\mathbb{R} \multimap (\mathrm{Staged}\, c \to \mathrm{Staged}\, c), \mathbb{R}))$$

Interleave changes to number all the scalars in the input with distinct IDs (for example with the same IDs as their corresponding input backpropagators, but this is not required); the cotangent of the input scalar with ID i is stored in the Map at key i. The input backpropagators can then modify the correct scalar in the collector (now of type $\mathsf{Map\ Int}\ \mathbb{R}$) in time logarithmic in the size of the input. To be able to construct the final gradient from this collection of just its scalars, $\mathrm{Interleave}_\tau$ can additionally build a *reconstruction* function of type $(\mathsf{Int} \to \mathbb{R}) \to \tau$, which we pass a function that looks up the ID in the final collector Map to compute the actual gradient value.[9]

Complexity. Now that we have fixed (in Section 5) the first complexity problem identified in Section 4.1 (expensive monoid operations) and reduced the second (expensive input backpropagators) to a logarithmic overhead over the original program, we have reached the point where we satisfy the complexity requirement stated in Section 4.1 apart from log-factors. More precisely, $\mathrm{Wrap}[P]$ computes the gradient of P at x not in time $O(\mathrm{cost}(P\ x) + \mathrm{size}(x))$ but in time $O((\mathrm{cost}(P\ x) + \mathrm{size}(x))\log(\mathrm{cost}(P\ x) + \mathrm{size}(x)))$. If we are okay with logarithmic overhead, we can stop here: the algorithm is already very close to efficient. However, if we wish to strictly conform to the required complexity, we need to make the input backpropagators and Map operations in SResolve constant-time; we do this using mutable arrays in Section 7.

[8] If c contains coproducts (sum types), this $\underline{0} : c$ becomes dependent on the actual input to the program, copying the structure from there.

[9] For an example implementation of this idea, see the $\mathrm{IArray}\, \mathbb{R} \to \tau$ in the return type of $\mathrm{Interleave}^4$ in the code transformation using arrays in Appendix A in the preprint [Smeding and Vákár 2022b].

7 USING MUTABLE ARRAYS TO SHAVE OFF LOG FACTORS

The analysis in Section 4.1 showed that after Cayley-transform in Section 5, the strict complexity requirements are met if we make the input backpropagators constant-time and make SResolve not do non-constant extra work over the actual backpropagators that it must call. Luckily, in both cases the only component that is not constant-time is the interaction with one of the Maps in Staged c:

$$\text{Staged}\, c = (\text{Map Int}\, \mathbb{R}, \text{Map Int}\, (\mathbb{R} \multimap (\text{Staged}\, c \to \text{Staged}\, c), \mathbb{R}))$$

The input backpropagators perform (logarithmic-time) updates to the first Map (the cotangent collector), and SResolve reads, deletes and updates entries in the second Map (the staging map for recording delayed backpropagator calls). Both of these Maps are keyed by increasing, consecutive integers starting from 0, and are thus ideal candidates to be replaced by an array:

$$\text{Staged}\, c = (\text{Array}\quad \mathbb{R}, \text{Array}\quad (\mathbb{R} \multimap (\text{Staged}\, c \xrightarrow{R} \text{Staged}\, c), \mathbb{R}))$$

To allocate an array, one must know how large it should be. Fortunately, at the time when we allocate the initial Staged c value using 0_{Staged} in SResolve, the primal pass has already been executed and we know (from the output ID of Interleave) how many input scalars there are, and (from the output ID of the transformed program) how many backpropagators there are. Hence, the size of these arrays is indeed known when they are allocated; and while these arrays are large, the resulting space complexity is equal to the worst case for reverse AD in general.

To get any complexity improvements from replacing a Map with an Array (indeed, to not pessimise the algorithm completely!), the write operations to the arrays need to be done *mutably*. These write operations occur in two places: in the updater functions produced by backpropagators (Staged $c \to$ Staged c) and in SResolve. Hence, in these two places we need an effectful function type; options include a resource-linear function type, written $\xrightarrow{R}$ above (as e.g. available in Rust[10] and Haskell [Bernardy et al. 2018]) and a monad for local side-effects such as the ST monad in Haskell [Launchbury and Jones 1994] (where one would get Staged $c \to$ ST s () instead). Our implementation (Section 10) uses resource-linear types.[11]

Time complexity. We now satisfy all the requirements of the analysis in Section 4.1, and hence have the correct time complexity for reverse AD. In particular, let I denote the size of the input and T the runtime of the original program. We can observe the following:

- The number of operations performed by $\mathbf{D}_c^3[t]$ (with the improvements from Sections 6 and 7) is only a constant factor times the number of operations performed by t, and hence in $O(T)$. This was already observed for $\mathbf{D}_c^2[t]$ in Section 4.1, and still holds.
- The number of backpropagators created while executing $\mathbf{D}_c^3[t]$ is clearly also in $O(T)$.
- The number of operations performed in any one backpropagator is constant. This is new, and only true because id (replacing 0_{Staged}), (∘) (replacing $(+_{\text{Staged}})$), SCotan (with a constant-time mutable array updater as argument) and SCall are now all constant-time.
- Hence, because every backpropagator is invoked at most once, and because the overhead of SResolve is constant per invoked backpropagator, the amount of work performed by calling the top-level input backpropagator is again in $O(T)$.
- Finally, the (non-constant-time) extra work performed in Wrap^3 is interleaving ($O(I)$), deinterleaving (O(size of output) and hence $O(T + I)$), resolving ($O(T)$) and reconstructing the gradient from the scalars in the Array $\mathbb{R}$ in Staged c ($O(I)$); all this work is in $O(T + I)$.

Hence, calling Wrap$^3[t]$ with an argument and calling its returned top-level derivative once takes time $O(T + I)$, i.e. at most proportional to the runtime of calling t with the same argument, plus

[10] https://www.rust-lang.org

[11] For more detail, see Appendix A in the preprint [Smeding and Vákár 2022b].

the size of the argument itself. This is indeed the correct time complexity for an efficient reverse AD algorithm, as discussed in Section 4.1.

Space complexity. The sizes of the allocated arrays are bounded by the total number of IDs we have generated, which equals the sum of the number of input scalars and the number of primitive operations executed at runtime in the original computation. This space complexity is standard for reverse AD. (Note that this may be different from the number of ground variables in the program.)

8 WAS IT TAPING ALL ALONG?

In this section we first apply one more optimisation to our algorithm to improve its complexity by a constant factor (Section 8.1). Next, we show that defunctionalising the backpropagators (Section 8.2) essentially reduces the technique to classical taping approaches (Section 8.3).

8.1 Dropping the Cotangent Collection Array

Recall that the final transformation of Section 7 used two mutable arrays threaded through the backpropagators in the Staged c pair: a cotangent collection array of type Array $\mathbb{R}$ and a backpropagator call staging array of type Array $(\mathbb{R} \multimap (\text{Staged}\, c \xrightarrow{R} \text{Staged}\, c), \mathbb{R})$. The first array is modified by $\text{Interleave}_{\mathbb{R}}$, and the second by SCall. No other functions modify these arrays.

Looking at the function of Interleave_R in the algorithm, all it does is produce input backpropagators with some ID i, which act by adding their argument to index i in the cotangent collection array. This means that if (c, m) is the input to SResolve for which the recursion terminates, we have $c[i] = \text{snd}(m[i])$ for all i for which $c[i]$ is defined. Therefore, the cotangent collection array is actually unnecessary: its information is directly readable from the backpropagator staging array.

With this knowledge, we can instead use Staged c = Array $(\mathbb{R} \multimap (\text{Staged}\, c \xrightarrow{R} \text{Staged}\, c), \mathbb{R})$ as our definition. The reconstruction functions of Section 6 simply take the second projection of the correponding array element.

8.2 Defunctionalisation of Backpropagators

In the core code transformation ($\mathbf{D}_c$, excluding the wrapper), all backpropagators are (now) of type $\mathbb{R} \multimap (\text{Staged}\, c \xrightarrow{R} \text{Staged}\, c)$, and, as observed earlier in Section 4, these backpropagators come in only a limited number of forms:

(1) the input backpropagators, as created in $\text{Interleave}_{\mathbb{R}}$, reduced to $(\underline{\lambda}(z : \mathbb{R}).\ \text{id})$ in Section 8.1;
(2) $(\underline{\lambda}(z : \mathbb{R}).\ \text{id})$, as created in $\mathbf{D}_c[r]$ for scalar constants r;
(3) $(\underline{\lambda}(z : \mathbb{R}).\ \text{SCall}\ d_1\ (\partial_1 op(x_1, \ldots, x_n)(z)) \circ \cdots \circ \text{SCall}\ d_n\ (\partial_n op(x_1, \ldots, x_n)(z)))$, as created in $\mathbf{D}_c[op(x_1, \ldots, x_n)]$ for primitive operations op.

Furthermore, the information contained in an operator backpropagator of form (3) can actually be described without reference to the value of its argument z: because our operators return a single scalar (as opposed to e.g. a vector), we have $\frac{\partial f(op(x_1,\ldots,x_n))}{\partial x_i} = \frac{\partial f(u)}{\partial u} \cdot \frac{\partial op(x_1,\ldots,x_n)}{\partial x_i}$, which can also be written as $\partial_i op(x_1, \ldots, x_n)(z) = z \cdot \partial_i op(x_1, \ldots, x_n)(1)$.

Hence, we can defunctionalise [Reynolds 1998] and change all occurrences of the type $\mathbb{R} \multimap (\text{Staged}\, c \xrightarrow{R} \text{Staged}\, c)$ to Contrib, where Contrib = $[(\mathbb{R}, (\text{Int}, \text{Contrib}))]$: a list of triples of a scalar, an integer ID, and a recursive Contrib structure. (The recursive Contrib structures are interpreted as having sharing as encoded by their IDs, similarly to how the references to existing backpropagators in the closures of operator backpropagators (3) already had sharing.) The meaning of $[(a_1, (i_1, cb_1)), \ldots, (a_n, (i_n, cb_n))]$ of type Contrib would then be:

$$\underline{\lambda}(z : \mathbb{R}).\ \text{SCall}\ (i_1, cb_1)\ (z \cdot a_1) \circ \cdots \circ \text{SCall}\ (i_n, cb_n)\ (z \cdot a_n)$$

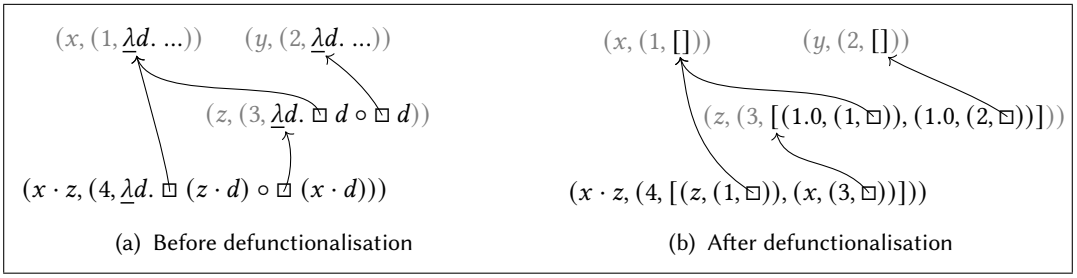


Fig. 10. The sharing structure before and after defunctionalisation. SCall is elided here; in Fig. 10a, the backpropagator calls are depicted as if they are still normal calls. Boxes (□) are the same in-memory value as the value their arrow points to; two boxes pointing to the same value indicates that this value is *shared*: referenced in two places.

For example, suppose we differentiate the following simple program:

$$\lambda(x, y).\ \textbf{let}\ z = x + y\ \textbf{in}\ x \cdot z$$

using the final algorithm of Section 7. The return value from the $\mathbf{D}_{\mathbb{R}}$-transformed code (when applied to the output from $\text{Interleave}_{\mathbb{R}}$) has the sharing structure shown in Fig. 10a. This shows how the backpropagators refer to each other in their closures.

If we perform the type replacement and the defunctionalisation, Interleave simplifies and SCall disappears, backpropagators of forms (1) and (2) become [] (the empty list) and those of form (3) become:

$$[(\text{fst}(d_1), (\text{snd}(d_1), \partial_1 op(x_1, \ldots, x_n)(1))), \ldots, (\text{fst}(d_n), (\text{snd}(d_n), \partial_n op(x_1, \ldots, x_n)(1)))]$$

SResolve then interprets a list of such $(i, (cb, a))$ by iterating over the list and for each such triple, replacing (cb', a') at index i in the staging array with $(cb, a' + a)$.

8.3 Was It Taping All Along?

After the improvements from Sections 8.1 and 8.2, what previously was a tree of (staged) calls to backpropagator functions is now a tree of Contrib values with attached IDs[12] that are interpreted by SResolve. This interpretation (eventually) writes the Contrib value with ID i to index i in the staging array (possibly multiple times), and furthermore accumulates argument cotangents in the second component of the pairs in the staging array. While the argument cotangents must be accumulated in reverse order of program execution (indeed, that is the whole point of *reverse* AD), the mapping from ID to Contrib value can be fully known in the forward pass: the partial derivatives of operators, $\partial_i op(x_1, \ldots, x_n)(1)$, can be computed in the forward pass already.

This means that if we change the ID generation monad that the differentiated code already lives in (which is a state monad with a single Int as state) to additionally carry the staging array, and furthermore change the monad to thread its state through resource-linearly,[13] we can already compute the Contrib lists and write them to the array in the forward pass. All that SResolve then has to do is loop over the array in reverse order (as it already does) and add cotangent contributions to the correct positions in the array according to the Contrib lists that it finds there.

At this point, there is no meaningful difference any more between this algorithm and what is classically known as taping: we have a tape (the staging array) that we write the performed operations to in the forward pass (automatically growing the array as necessary)—although the tape entries are the already-differentiated operations in this case, and not the original ones. In this way, we have related the naive version of dual-numbers reverse AD, which admits neat correctness

[12]Note that we now have $\mathbf{D}[\![\mathbb{R}]\!] = (\mathbb{R}, (\text{Int}, \text{Contrib}))$, the integer being the ID of the Contrib value.
[13]This is possible and results in a linear variant of standard Haskell monads, as described in [Bernardy et al. 2018].

proofs, to the classical, very imperative approach to reverse AD based on taping, which is used in industry-standard implementations of reverse AD (e.g. PyTorch [Paszke et al. 2017]).

9 EXTENDING THE SOURCE LANGUAGE

The source language (Fig. 4) that the algorithm discussed so far works on, is a higher-order functional language including product types and primitive operations on scalars. However, dual-numbers reverse AD generalises to much richer languages in a very natural way, because most of the interesting work happens in the scalar primitive operations. The efficiency of the algorithm is independent of the language constructs in the source language. Indeed, in the forward pass, the code transformation is fully structure-preserving outside of the scalar constant and primitive operation cases; and in the reverse pass (in SResolve), all program structure is forgotten anyway, because the computation is flattened to a linear sequence of primitive operations on scalars.

(Mutual) recursion. For example, we can allow recursive functions in our source language by adding the syntax $\mathbf{letrec}\ f : \sigma \to \tau = \lambda(x : \sigma).\ s\ \mathbf{in}\ t$. The code transformation $\mathbf{D}^i$ for all i then treats **letrec** exactly the same as **let**—note that the only syntactic difference between **letrec** and **let** is the scoping of f—and the algorithm remains both correct and efficient.

Coproducts. To support dynamic control flow (necessary to make recursion useful), we can easily add coproducts to the source language. First add coproducts to the syntax for types ($\sigma, \tau ::= \cdots \mid \sigma + \tau$) both in the source language and in the target language, and add constructors and eliminators to all term languages (both linear and non-linear):

$$s, t ::= \cdots \mid \mathrm{inl}(t) \mid \mathrm{inr}(t) \mid \mathbf{case}\ s\ \mathbf{of}\ \{\mathrm{inl}(x) \to t_1; \mathrm{inr}(y) \to t_2\}$$

where x and y are in scope in t_1 and t_2. Then the type and code transformations extend in the unique structure-preserving manner:

$$\mathbf{D}_c^1[\sigma + \tau] = \mathbf{D}_c^1[\sigma] + \mathbf{D}_c^1[\tau]$$
$$\mathbf{D}_c^1[\mathrm{inl}(t)] = \mathrm{inl}(\mathbf{D}_c^1[t]) \qquad \mathbf{D}_c^1[\mathrm{inr}(t)] = \mathrm{inr}(\mathbf{D}_c^1[t])$$
$$\mathbf{D}_c^1[\mathbf{case}\ s\ \mathbf{of}\ \{\mathrm{inl}(x) \to t_1; \mathrm{inr}(x) \to t_2\}] = \mathbf{case}\ \mathbf{D}_c^1[s]\ \mathbf{of}\ \{\mathrm{inl}(x) \to \mathbf{D}_c^1[t_1]; \mathrm{inr}(x) \to \mathbf{D}_c^1[t_2]\}$$

The type transformation stays unchanged when moving to $\mathbf{D}_c^3$, and the only change for the term definitions is to transition to monadic code in $\mathbf{D}_c^2$. Lifting a computation to monadic code is a well-understood process. The corresponding cases in Interleave and Deinterleave are the only reasonable definitions that type-check.

Polymorphic and (mutually) recursive types. In Haskell one can define (mutually) recursive data types e.g. as follows:

$$\mathbf{data}\ T_1\ \alpha = C_1\ \alpha\ (T_2\ \alpha) \mid C_2\ \mathbb{R}$$
$$\mathbf{data}\ T_2\ \alpha = C_3\ \mathsf{Int}\ (T_1\ \alpha)\ (T_2\ \alpha).$$

If the user has defined some data types, then we can allow these data types in the code transformation. We add new *data type declarations* that simply apply $\mathbf{D}_c^1[-]$ to all parameter types of all constructors:

$$\mathbf{data}\ DT_1\ \alpha = DC_1\ \alpha\ (DT_2\ \alpha) \mid DC_2\ (\mathbb{R}, \mathbb{R} \multimap c)$$
$$\mathbf{data}\ DT_2\ \alpha = DC_3\ \mathsf{Int}\ (DT_1\ \alpha)\ (DT_2\ \alpha).$$

and we add one rule for each data type that simply maps:[14]

$$\mathbf{D}_c^1[T_1\ \tau] = DT_1\ \mathbf{D}_c^1[\tau] \qquad \mathbf{D}_c^1[T_2\ \tau] = DT_2\ \mathbf{D}_c^1[\tau].$$

Furthermore, for plain type variables, we set $\mathbf{D}_c^1[\alpha] = \alpha$.

The code transformation on terms is completely analogous to a combination of coproducts (given above in this section, where we take care to match up constructors as one would expect: C_i gets sent to DC_i) and products (given already in Fig. 6). The wrapper also changes analogously: Interleave and Deinterleave get clauses for $\text{Interleave}_{(T_i\ \tau)}$ and $\text{Deinterleave}_{(T_i\ \tau)}$.

Finally, we note that with the mentioned additional rule that $\mathbf{D}_c^1[\alpha] = \alpha$, polymorphic functions can also be differentiated transparently, similarly to how the above handles polymorphic data types.

10 IMPLEMENTATION

To show the practicality of our method, we provide a prototype implementation[15] of the resulting algorithm of Section 7, together with the improvements from Sections 8.1 and 8.2, that differentiates a sizeable fragment of Haskell98 including recursive types (reinterpreted as a strict, call-by-value language) using Template Haskell. We realise the mutable arrays of Section 7 using resource-linearly typed arrays of Linear Haskell [Bernardy et al. 2018], which are similar in intent, though not identical in design, to those of the Rust language. The implementation does not incorporate the changes given in Section 8.3 that transform the algorithm into classical taping, but it does include support for recursive functions and user-defined data types as described in Section 9.

Template Haskell [Sheard and Jones 2002] is a built-in metaprogramming facility in GHC Haskell that (roughly) allows the programmer to write a Haskell function that takes a block of user-written Haskell code, do whatever it wants with the AST of that code, and finally splice the result back into the user's program. The resulting code is still type-checked as usual. The AST transformation that we implement is, of course, differentiation.

Benchmarks. To check that our implementation has reasonable performance in practice, we benchmark (in `bench/Main.hs`) against Kmett's `ad` library [Kmett and contributors 2021] (version 4.5) on a few basic functions. These functions are:

- A single scalar multiplication of type `(Double, Double) -> Double`;
- Dot product of type `([Double], [Double]) -> Double`;
- Matrix–vector multiplication, then sum: of type `([[Double]], [Double]) -> Double`;
- The `rotate_vec_by_quat` example from [Krawiec et al. 2022] of type `(Vec3 Double, Quaternion Double) -> Vec3 Double`, with `data Vec3 s = Vec3 s s s` and `data Quaternion s = Quaternion s s s s`.

The last case has a non-trivial return type.

The benchmark results are summarised in Table 1. (For the `criterion` report showing linear complexity scaling, see Appendix B in the preprint [Smeding and Vákár 2022b].) The benchmarks are timed using the `criterion`[16] library. To get statistically significant results, we measure how the timings scale with increasing n:

- Scalar multiplication and `rotate_vec_by_quat` are simply differentiated n times;
- Dot product is performed on lists of length n;
- Matrix multiplication is done for a matrix and vector of size $\sqrt{n}$, to get linear scaling in n.

[14] As declaring new data types is inconvenient in Template Haskell, our current implementation only handles recursive data types that do not contain explicit scalar values. As we can pass all required scalar types by instantiating their type parameters with a type containing $\mathbb{R}$, this is not a real restriction.

[15] The code is available at https://github.com/tomsmeding/ad-dualrev-th, archived at [Smeding and Vákár 2022a].

[16] By Bryan O'Sullivan: https://hackage.haskell.org/package/criterion

Table 1. Benchmark results of Section 7 + Sections 8.1 and 8.2 versus ad-4.5. The 'TH' and 'ad' columns indicate runtimes on one machine for our implementation and the ad library, respectively. The last column shows the ratio between the previous two columns. We give the size of the largest side of `criterion`'s 95% confidence interval. Setup: GHC 9.2.2 on Linux, Intel i9-10900K CPU. Benchmarks are single-threaded.

	TH	ad	TH / ad
scalar mult.	0.146 μs ±0.000	0.536 μs ±0.002	0.27
dot product	2.21 μs ±0.10	2.07 μs ±0.06	≈1.1
sum-mat-vec	2.05 μs ±0.14	1.32 μs ±0.05	≈1.5
rotate_vec_by_quat	8.77 μs ±0.01	6.13 μs ±0.02	≈1.43

By the results in Table 1, we see that for less trivial programs, our implementation has reasonable constant-factor performance compared to the highly-optimised ad library. Note that our goal here is merely to substantiate the claim that the implementation exhibits constant-factor performance in the right ballpark (in addition to it having the right asymptotic complexity, as we have argued). Our benchmarks include key components of many AD applications, such as neural nets, and seeing that we have not at all special-cased their implementation, we believe that they suffice to demonstrate our limited claim. As our code is simply a proof-of-concept with minimal effort spent on performance, we conclude from this that the algorithm described in this paper indeed admits a work-efficient implementation. We leave further optimisation of our implementation and extensive benchmarking to future work.

11 CONCLUSIONS

One may ask: if the final algorithm from Section 7 can be argued to be "just taping" (Section 8.3), which is already widely used in practice, what was the point? The point is the observation that optimisations are key to implementing efficient AD and that multiple kinds of reverse AD algorithms (in particular the one from Fig. 6, studied in [Brunel et al. 2020] and [Huot et al. 2020, Section 6], but further examples are in Section 12.2) tend to all reduce to taping after optimisation. However, algorithms outside this category of course also exist: we believe that CHAD [Vákár and Smeding 2022] is genuinely different and does not lead to a sequentialised dependency graph or linear "tape".

The first of our optimisations (linear factoring) is quite specific to starting AD algorithms that need some kind of distributive law to become efficient (e.g. also [Krawiec et al. 2022]). However, we think that the other optimisations are more widely applicable (and will, for example, also be key steps to making CHAD efficient): sparse vectors will probably be needed in most functional reverse AD algorithms to efficiently represent the one-hot vectors resulting from projections (`fst`/`snd` as well as random access into arrays, through indexing), and mutable arrays are a standard solution to remove the ubiquitous log-factor in the complexity of purely functional algorithms.

12 ORIGINS OF DUAL-NUMBERS REVERSE AD, RELATIONSHIP WITH VECTORISED FORWARD AD AND OTHER RELATED WORK

The literature about automatic differentiation spans many decades and academic subcommunities (scientific computing, machine learning and—most recently—programming languages). Important early references are [Linnainmaa 1970; Speelpenning 1980; Wengert 1964]. Good surveys can be found in [Baydin et al. 2017; Margossian 2019]. In the rest of this section, we focus on the more recent literature that studies AD from a programming languages (PL) point of view, to extend the scope of our discussion of Section 8.

12.1 Theoretical Foundations for Our Algorithm

The first mention that we know of the naive dual-numbers reverse mode AD algorithm that we analyse in this paper is [Pearlmutter and Siskind 2008, page 12], where it is quickly dismissed before a different technique is pursued. The algorithm is first thoroughly studied by [Brunel et al. 2020] using operational semantics and [Huot et al. 2020, Section 6] using denotational semantics. [Brunel et al. 2020] introduces the key idea that underlies the efficient implementation of our paper: the linear factoring rule, stating that a term $f\ x + f\ y$, with f a linear function, may be reduced to $f\ (x+y)$. We build on their use of this rule as a tool in a complexity proof to make it a suitable basis for a performant implementation. We achieve this by noting that it can be efficiently implemented using our Staged c data structure combined with runtime numbering of backpropagators and next observing that we obtain a performant implementation if we apply a Cayley transformation (and use mutable arrays to shave off log-factors).

[Mazza and Pagani 2021] extends the work of [Brunel et al. 2020] to apply to a language with term recursion, showing that dual-numbers reverse AD on PCF is almost everywhere correct. Similarly, [Nunes and Vákár 2022a,b] extend the work of [Huot et al. 2020] to apply to partial programs involving iteration, recursion and recursive types, thus giving a correctness proof for the initial dual-numbers reverse AD transformation of Fig. 6 applied to idealised Haskell98.

12.2 Vectorised Forward AD

Furthermore, there are strong parallels with the derivation in [Krawiec et al. 2022]. Like the present paper, they give a sequence of steps that optimise a simple algorithm to an efficient implementation—but the starting algorithm is vectorised forward AD (VFAD) instead of backpropagator-based dual-numbers reverse AD (DNRAD). In our notation, their initial type transformation does not have $\mathbf{D}^1_c[\mathbb{R}] = (\mathbb{R}, \mathbb{R} \multimap c)$, but instead $\mathbf{D}^1_c[\mathbb{R}] = (\mathbb{R}, c)$. (As befits a dual-numbers algorithm, the rest of the type transformation is simply structurally recursive.)

Linear algebra tells us that the vector spaces $\mathbb{R} \multimap c$ and c are isomorphic, and indeed inspection of the term transformations shows that both naive algorithms compute the same thing. Their operational behaviour, on the other hand, is very different: the complexity problem with DNRAD is exponential blowup in the presence of sharing, whereas VFAD is "simply" n times too slow, where n is the number of scalars in the input.

But the first optimisation on VFAD, which defunctionalises the zero, one-hot, addition and scaling operations on the c tangent vector, introduces the same sharing-induced complexity problem as we have in naive DNRAD as payment for fixing the factor-n overhead. The two algorithms are now on equal footing: we could defunctionalise the backpropagators in DNRAD just as easily.

Afterwards, VFAD is lifted to a combination (stack) of an ID-generation monad and a Writer monad. Each scalar result of a primitive operation gets an ID, and the Writer monad records for each ID (hence, scalar in the program) its defunctionalised tangent vector (i.e. an expression) in terms of other already-computed tangent vectors from the Writer record. These expressions correspond to our primitive operation backpropagators with calls replaced with SCall: where we replace calls with ID-tagged pairs of function and argument, VFAD replaces the usage of already-computed tangent vectors with scaled references to the IDs of those vectors. The choice in our SResolve of evaluation order from highest ID to lowest ID (Section 4) is encoded in VFAD's definitions of *runDelta* and *eval*, which process the record back-to-front.

Finally, our Cayley-transform is encoded in the type of VFAD's *eval* function, which interprets the defunctionalised operations on tangent vectors (including explicit sharing using the Writer log) into an actual tangent vector—the final gradient: its gradient return type is Cayley-transformed.

Our final optimisation to mutable arrays to eliminate log-factors in the complexity is also mirrored in VFAD.

Distributive law. Under the isomorphism $\mathbb{R} \multimap c \cong c$, the type Staged c can be thought of as a type Expr c of ASTs of expressions (with sharing) of type c.[17] The linear factoring rule $f\ x{+}f\ y \rightsquigarrow f\ (x{+}y)$ for a linear function $f : \mathbb{R} \multimap c$ that rescales a vector $v : c$ with a scalar then corresponds to the distributive law $v \cdot x + v \cdot y \rightsquigarrow v \cdot (x + y)$. This highlights the relationship between our work and that of [Shaikhha et al. 2019], which tries to (statically) optimise vectorised forward AD to reverse AD using precisely this distributive law. A key distinction is that we apply this law (in the form of the linear factoring rule) at runtime rather than compile time, allowing us to always achieve the complexity of reverse AD, rather than merely on some specific programs with straightforward control and data flow. The price we pay for this generality is a runtime overhead, similar to the usual overhead of taping.

Getting IDs from memory addresses. One final possible optimisation to our implementation that we expect to improve its constant-factor performance is to extract the IDs for scalars from the identity of their heap object (in a sense: their memory address) whenever an ID is used. This would allow the code transformation to work without the use of an ID generation monad. In Haskell, this is possible, for example, by looking up the StableName corresponding to a scalar at runtime. [Kmett and contributors 2021] follows this strategy, and we expect that this explains much of the difference in constant-factor performance between our implementation and theirs.

12.3 Other PL Literature About AD

CHAD and category theory inspired AD. Rather than interleaving backpropagators by pairing them with scalars in a type, we can also try to directly implement reverse AD as a structurally recursive code transformation that does not need a (de)interleaving wrapper. This is the approach taken by Elliott [Elliott 2018]. It pairs vectors (and values of other composite types) with a single composite backpropagator, rather than decomposing to the point where each scalar is paired with a mini-backpropagator like in our dual-numbers approach. The resulting algorithm is extended to source languages with function types in [Vákár 2021; Vákár and Smeding 2022; Vytiniotis et al. 2019] and to sum and (co)inductive types in [Nunes and Vákár 2021]. Like our dual-numbers reverse AD approach, the algorithm arises as a canonical structure-preserving functor on the syntax of a programming language. However, due to a different choice in target category (a Grothendieck construction of a linear λ-calculus for CHAD rather than the syntax of a plain λ-calculus for dual-numbers AD), the resulting algorithm looks very different.

Approaches utilising non-local control flow. Another category of approaches to AD recently taken by the PL community are those that rely on forms of non-local control flow such as delimited continuations [Wang and Rompf 2018] or effect handlers [de Vilhena and Pottier 2021; Sigal 2021]. These techniques are different in the sense that they generate code that is not purely functional. This use of non-local control flow makes it possible to achieve an efficient implementation of reverse AD that looks strikingly simple compared to alternative approaches. Where the CHAD approaches and our dual-numbers reverse AD approach both have to manually invert control flow at compile time by making use of continuations that get passed around, combined with smart staging of execution of those continuations in our case, this inversion of control can be deferred to run time by clever use of delimited control operators or effect handlers.

[17] In [Krawiec et al. 2022], Expr c is called Delta.

REFERENCES

Martín Abadi and Gordon D. Plotkin. 2020. A simple differentiable programming language. *Proc. ACM Program. Lang.* 4, POPL (2020), 38:1–38:28. https://doi.org/10.1145/3371106

Atilim Gunes Baydin, Barak A. Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. 2017. Automatic Differentiation in Machine Learning: a Survey. *J. Mach. Learn. Res.* 18 (2017), 153:1–153:43. http://jmlr.org/papers/v18/17-468.html

Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R. Newton, Simon Peyton Jones, and Arnaud Spiwack. 2018. Linear Haskell: practical linearity in a higher-order polymorphic language. *Proc. ACM Program. Lang.* 2, POPL (2018), 5:1–5:29. https://doi.org/10.1145/3158093

Aloïs Brunel, Damiano Mazza, and Michele Pagani. 2020. Backpropagation in the simply typed lambda-calculus with linear negation. *Proc. ACM Program. Lang.* 4, POPL (2020), 64:1–64:27. https://doi.org/10.1145/3371132

Paulo Emílio de Vilhena and François Pottier. 2021. Verifying a Minimalist Reverse-Mode AD Library. *arXiv preprint arXiv:2112.07292* (2021).

Conal Elliott. 2018. The simple essence of automatic differentiation. *Proc. ACM Program. Lang.* 2, ICFP (2018), 70:1–70:29. https://doi.org/10.1145/3236765

Andreas Griewank and Andrea Walther. 2008. *Evaluating derivatives - principles and techniques of algorithmic differentiation, Second Edition.* SIAM. https://doi.org/10.1137/1.9780898717761

R. John M. Hughes. 1986. A Novel Representation of Lists and its Application to the Function "reverse". *Inf. Process. Lett.* 22, 3 (1986), 141–144. https://doi.org/10.1016/0020-0190(86)90059-1

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

Edward Kmett and contributors. 2021. *ad: Automatic Differentiation.* https://hackage.haskell.org/package/ad

Faustyna Krawiec, Simon Peyton Jones, Neel Krishnaswami, Tom Ellis, Richard A. Eisenberg, and Andrew W. Fitzgibbon. 2022. Provably correct, asymptotically efficient, higher-order reverse-mode automatic differentiation. *Proc. ACM Program. Lang.* 6, POPL (2022), 1–30. https://doi.org/10.1145/3498710

John Launchbury and Simon L. Peyton Jones. 1994. Lazy Functional State Threads. In *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation (PLDI), Orlando, Florida, USA, June 20-24, 1994*, Vivek Sarkar, Barbara G. Ryder, and Mary Lou Soffa (Eds.). ACM, 24–35. https://doi.org/10.1145/178243.178246

Seppo Linnainmaa. 1970. The representation of the cumulative rounding error of an algorithm as a Taylor expansion of the local rounding errors. *Master's Thesis (in Finnish), Univ. Helsinki* (1970).

Charles C. Margossian. 2019. A review of automatic differentiation and its efficient implementation. *Wiley Interdiscip. Rev. Data Min. Knowl. Discov.* 9, 4 (2019). https://doi.org/10.1002/widm.1305

Damiano Mazza and Michele Pagani. 2021. Automatic differentiation in PCF. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–27. https://doi.org/10.1145/3434309

Fernando Lucatelli Nunes and Matthijs Vákár. 2021. CHAD for Expressive Total Languages. *CoRR* abs/2110.00446 (2021). arXiv:2110.00446 https://arxiv.org/abs/2110.00446

Fernando Lucatelli Nunes and Matthijs Vákár. 2022a. Automatic Differentiation for ML-family languages: correctness via logical relations. *CoRR* abs/2210.07724 (2022). arXiv:2210.07724 https://arxiv.org/abs/2210.07724

Fernando Lucatelli Nunes and Matthijs Vákár. 2022b. Logical Relations for Partial Features and Automatic Differentiation Correctness. *CoRR* abs/2210.08530 (2022). arXiv:2210.08530 https://arxiv.org/abs/2210.08530

Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in PyTorch. In *NIPS 2017 Autodiff Workshop: The future of gradient-based machine learning software and techniques.* Curran Associates, Inc., Red Hook, NY, USA.

Adam Paszke, Daniel Johnson, David Duvenaud, Dimitrios Vytiniotis, Alexey Radul, Matthew Johnson, Jonathan Ragan-Kelley, and Dougal Maclaurin. 2021. Getting to the Point. Index Sets and Parallelism-Preserving Autodiff for Pointful Array Programming. *CoRR* abs/2104.05372 (2021). arXiv:2104.05372 [cs.PL]

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

John C. Reynolds. 1998. Definitional Interpreters for Higher-Order Programming Languages. *High. Order Symb. Comput.* 11, 4 (1998), 363–397. https://doi.org/10.1023/A:1010027404223

Robert Schenck, Ola Rønning, Troels Henriksen, and Cosmin E. Oancea. 2022. AD for an Array Language with Nested Parallelism. *CoRR* abs/2202.10297 (2022). arXiv:2202.10297 https://arxiv.org/abs/2202.10297

Amir Shaikhha, Andrew Fitzgibbon, Dimitrios Vytiniotis, and Simon Peyton Jones. 2019. Efficient differentiable programming in a functional array-processing language. *Proc. ACM Program. Lang.* 3, ICFP (2019), 97:1–97:30. https://doi.org/10.1145/

3341701

Tim Sheard and Simon L. Peyton Jones. 2002. Template meta-programming for Haskell. *ACM SIGPLAN Notices* 37, 12 (2002), 60–75. https://doi.org/10.1145/636517.636528

Jesse Sigal. 2021. Automatic differentiation via effects and handlers: An implementation in Frank. *arXiv preprint arXiv:2101.08095* (2021).

Tom Smeding and Matthijs Vákár. 2022a. Artifact for Efficient Dual-Numbers Reverse AD via Well-Known Program Transformations. https://doi.org/10.5281/zenodo.7130343 Artifact for this publication.

Tom Smeding and Matthijs Vákár. 2022b. Efficient Dual-Numbers Reverse AD via Well-Known Program Transformations. *CoRR* abs/2207.03418v2 (2022). https://doi.org/10.48550/arXiv.2207.03418 arXiv:2207.03418v2

B. Speelpenning. 1980. *Compiling fast partial derivatives of functions given by algorithms.* Technical Report. Illinois University. https://doi.org/10.2172/5254402

Matthijs Vákár. 2021. Reverse AD at Higher Types: Pure, Principled and Denotationally Correct. In *Programming Languages and Systems (Lecture Notes in Computer Science, Vol. 12648)*, Nobuko Yoshida (Ed.). Springer, 607–634. https://doi.org/10.1007/978-3-030-72019-3_22

Matthijs Vákár and Tom Smeding. 2022. CHAD: Combinatory Homomorphic Automatic Differentiation. *ACM Trans. Program. Lang. Syst.* 44, 3, 20:1–20:49. https://doi.org/10.1145/3527634

Dimitrios Vytiniotis, Dan Belov, Richard Wei, Gordon Plotkin, and Martin Abadi. 2019. The differentiable curry. *NeurIPS Workshop on Program Transformations* (2019).

Fei Wang and Tiark Rompf. 2018. A Language and Compiler View on Differentiable Programming. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Workshop Track Proceedings*. OpenReview.net. https://openreview.net/forum?id=SJxJtYkPG

R. E. Wengert. 1964. A simple automatic derivative evaluation program. *Commun. ACM* 7, 8 (1964), 463–464. https://doi.org/10.1145/355586.364791

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