JFP **35**, e16, 66 pages, 2025.

doi:10.1017/S0956796825100051

Parallel dual-numbers reverse AD

TOM J. SMEDING
Utrecht University, The Netherlands
(*e-mail:* t.j.smeding@uu.nl)

MATTHIJS I. L. VÁKÁR
Utrecht University, The Netherlands
(*e-mail:* m.i.l.vakar@uu.nl)

Abstract

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. Where previous work on dual numbers reverse AD has required sequentialisation to construct the reverse pass, we demonstrate that we can apply our technique to task-parallel source programs and generate a task-parallel derivative computation.

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, locally) 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 it 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 the desired computational complexity. Indeed, reverse AD computes the gradient of a function implemented as a program in time[1] at most a constant factor more than the 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 flexible programming languages: 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*a*), or sometimes specialised by taking advantage of common usage patterns in domain-specific languages (Schenck et al., 2022). In the programming language 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). Although 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, theoretical dual-numbers reverse AD algorithm analysed by Brunel et al. (2020) to a practical functional taping approach as used by Kmett and Contributors (2021) and analysed by Krawiec et al. (2022). Further, we ensure that the implementation exploits parallelism opportunities for the derivative computation that arise from parallelism present in the source program, rather than sequentialising the derivative computation, as was done by Kmett and Contributors (2021); Krawiec et al. (2022); Smeding and Vákár (2023). The key point made by Brunel, Mazza, and Pagani (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 5), 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 $f\ (x + y)$. We demonstrate how this factoring idea can motivate an efficient practical implementation that is also parallelism-preserving.

Summary of contributions. 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 6.
- We show how this algorithm can be made complexity efficient by using the standard functional programming techniques of Cayley transformation (Section 7) and (e.g. linearly typed or monadic) functional in-place updates (Section 9).

[1] This refers to the number of primitive operations (including memory operations, etc.); naturally, the actual wall-clock time depends also on cache behaviour, memory locality, etc., which we consider out of scope.

https://doi.org/10.1017/S0956796825100051 Published online by Cambridge University Press

- We explain how our algorithm relates to classical approaches based on taping (Section 10).
- We demonstrate that, by contrast with previous similar approaches (Kmett and Contributors, 2021; Krawiec et al., 2022; Smeding and Vákár, 2023), we do not need to sequentialise the derivative computation in case of a parallel source program, but instead can store the task parallelism structure during the primal pass and produce a task-parallel derivative (Section 12).
- We give an implementation of the parallelism-ready algorithm of Section 12 that can differentiate most of Haskell98 (but using call-by-value semantics) and that has the correct asymptotic complexity as well as decent constant-factor performance (Section 13).
- 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 15). We also briefly describe the broader relationship with related work.

Relation to previous work. This paper extends and develops our previous work (Smeding and Vákár, 2023) presented at the 50th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2023). This version includes multiple elaborations. Most notable is an analysis of how a variant of our dual numbers reverse AD algorithm can be applied to task-parallel source programs to produce task-parallel derivative code (Section 12) and an implementation of this parallelism-preserving AD method (`https://github.com/tomsmeding/ad-dualrev-th`). Besides this novel parallel AD technique, we include extra sections to explain our method better: a detailed discussion of the desired type of reverse AD (Section 4) including a comparison to CHAD (Vákár and Smeding, 2022; Nunes and Vákár, 2023; Smeding and Vákár, 2024) and a detailed description of how to use mutable arrays to eliminate the final log factors from the complexity of our method (Section 9).

2 Key ideas

Forward and reverse AD. All modes of automatic differentiation exploit the idea that derivatives satisfy the chain rule:

$$D_x(f \circ g) = D_{g(x)}(f) \cdot D_x(g)$$

where we write the derivative of $h : \mathbb{R}^n \to \mathbb{R}^m$ at the point $x \in \mathbb{R}^n$ as $D_x(h)$, a linear map $\underline{\mathbb{R}}^n \multimap \underline{\mathbb{R}}^m$ that computes directional derivatives.[2] In linear algebra, this linear map corresponds to the Jacobian matrix of partial derivatives of h: if $Jh(x)$ is the Jacobian of h at x, then $D_x(h)(v) = Jh(x) \cdot v$.

Using the chain rule, one can mechanically compute derivatives of a composition of functions. The intermediate function values (such as x and $g(x)$ in the example above)

[2] These are known as Fréchet derivatives (we write $\underline{\mathbb{R}}$ for the Euclidian vector space on the set $\mathbb{R}$). The function $\lambda x.\ h(p) + D_p(h)(x - p)$ is the best linear approximation to h around some point $p \in \mathbb{R}^n$. Equivalently: $D_p(h)$ describes how h's output changes given a small perturbation to its input, i.e. $h(p) + D_p(h)(\Delta p)$ is close to $h(p + \Delta p)$ for small values of Δp.

https://doi.org/10.1017/S0956796825100051 Published online by Cambridge University Press

are called *primals*; the (forward) derivative values (e.g. the input and output of $D_x(g)$ and $D_{g(x)}(f)$) are called *tangents*.

Forward AD directly implements the chain rule above as a code transformation to compute derivatives. Reverse AD instead computes the *transposed derivative*, which satisfies this *contravariant* chain rule instead:

$$D_x(f \circ g)^t = D_x(g)^t \cdot D_{g(x)}(f)^t$$

Here, we write $D_x(h)^t : \underline{\mathbb{R}}^m \multimap \underline{\mathbb{R}}^n$ for the transpose of the linear map $D_x(h) : \underline{\mathbb{R}}^n \multimap \underline{\mathbb{R}}^m$; in relation to the Jacobian of h, we have $D_x(h)^t(w) = w \cdot Jh(x)$. The values taken by and produced from transposed derivatives (or *reverse derivatives*), such as $D_x(g)^t$ and $D_{g(x)}(f)^t$ above, are called *cotangents* in this paper; other literature also uses the word *adjoint* for this purpose.[3]

Computing the full Jacobian of a function $f : \mathbb{R}^n \to \mathbb{R}^m$ at $x : \mathbb{R}^n$ requires n evaluations of $D_x(f) : \underline{\mathbb{R}}^n \multimap \underline{\mathbb{R}}^m$ (at the n basis vectors of $\underline{\mathbb{R}}^n$; one such evaluation computes one column of the Jacobian), or alternatively m evaluations of $D_x(f)^t : \underline{\mathbb{R}}^m \multimap \underline{\mathbb{R}}^n$ (one such evaluation computes one row of the Jacobian). Because many applications have $n \gg m$, reverse AD is typically of most interest, but forward AD is much simpler: for both modes (forward and reverse), we need the primals in the derivative computation, but in forward mode the derivatives are computed *in the same order* as the original computation (and hence as the computation of the primals). In reverse mode, the derivatives are computed in opposite order, requiring storage of all[4] primals during a *forward pass* of program execution, after which the *reverse pass* uses those stored primals to compute the derivatives.[5]

This observation that, in forward AD, primals and tangents can be computed in tandem, leads to the idea of *dual-numbers forward AD*: pair primals and derivatives explicitly to interleave the primal and derivative computations, and run the program with overloaded arithmetic operators to propagate these tangents forward using the chain rule. The tangent of the output can be read from the tangents paired up with the output scalars of the program. For example, transforming the program in Figure 1(a) using dual-numbers forward AD yields Figure 1(b). As a result, we do not store the primals any longer than is necessary, and the resulting code transformation tends to be much simpler than a naive attempt that first computes and stores all primals before computing any derivatives.

Naive dual-numbers reverse AD. For reverse AD, it is in general not possible to interleave the primal and derivative computations, as the reversal of derivatives generally requires us to have computed all primals before starting the derivative computation. However, by choosing a clever encoding, it is possible to make reverse AD *look* like a dual-numbers-style code transformation. Even if the primal and derivative computations will not be performed in an interleaved fashion, we can interleave the primal computation

[3] If we write $\nabla_x f$ for the *gradient* at x of a scalar-valued function f, then we can rewrite the Jacobian equality to $D_x(h)^t(w) = w \cdot (\nabla_x h_1, \ldots, \nabla_x h_n)$. In particular, for $n = 1$ if $w = (1)$, one clearly recovers h's gradient, which describes in what direction to move h's input to make its (scalar) output change the fastest, and how fast it changes in that case.

[4] Or nearly all, in any case, if optimal time complexity is desired. A trade-off between time and space complexity is provided by checkpointing (e.g. Siskind and Pearlmutter, 2018); a constant-factor improvement may be obtained by skipping some unnecessary primals (e.g. Hascoët et al., 2005).

[5] For a survey of AD in general, see e.g. Baydin et al. (2017).

https://doi.org/10.1017/S0956796825100051 Published online by Cambridge University Press

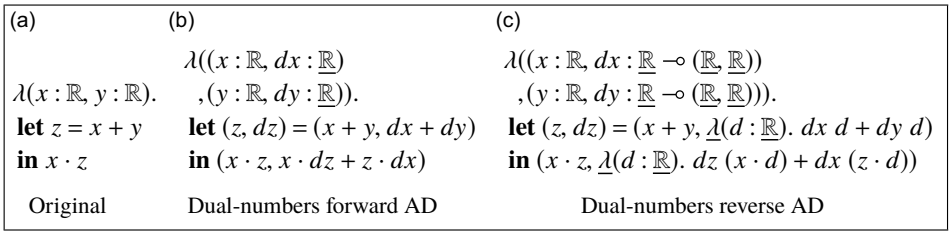


Fig. 1. An example program together with its derivative, both using dual-numbers forward AD and using dual-numbers reverse AD. The original program is of type $(\mathbb{R}, \mathbb{R}) \to \mathbb{R}$.

with a computation that *builds* a delayed reverse derivative function. Afterwards, we still need to call the derivative function that has been built. The advantage of such a "dual-numbers-style" approach to reverse AD is that the code transformation can be simple and widely applicable.

To make reverse AD in dual-numbers style possible, we have to encode the "reversal" in the tangent scalars that we called dx and dy in Figure 1(b). A solution is to replace those tangent scalars with *linear functions* that take the *cotangent* of the scalar it is paired with, and return the cotangent of the full input of the program. Transforming the same example program Figure 1(a) using this style of reverse AD yields Figure 1(c). The linearity indicated by the $\multimap$-arrow here is that of a monoid homomorphism (a function preserving 0 and $(+)$[6]); operationally, however, linear functions behave just like regular functions.

This naive dual-numbers reverse AD code transformation, shown in full in Figure 6, is simple and it is easy to see that it is correct via a logical relations argument (Lucatelli Nunes and Vákár, 2024). The idea of this argument is to prove via induction that a backpropagator $dx : \underline{\mathbb{R}} \multimap c$ that is paired with an intermediate value $x : \mathbb{R}$ in the program, computes the reverse (i.e. transposed) derivative of the subcomputation that calculates $x : \mathbb{R}$ from the global input to the program.[7] c is a type parameter of the code transformation, and represents the type of cotangents to the program input; if this input (of type τ, say) is built from just scalars, discrete types and product and sum types, as we will assume in this paper, it suffices (for correctness, not efficiency) to set $c = \tau$.[8]

Dual-numbers forward AD has the very useful property that it generalises over many types (e.g. products, coproducts, and recursive types) and program constructs (e.g. recursion, higher-order functions), thereby being applicable to many functional languages; 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), naive dual-numbers reverse AD is wildly inefficient: calling dx_n returned by the differentiated program in Figure 2 takes time *exponential* in n. Such overhead would make reverse AD completely

[6] And also scaling, making it a vector space homomorphism. The vector space structure of tangents and cotangents tends to be unused in AD.

[7] That is to say: "dx 1" returns the gradient of x with respect to the program input.

[8] In fact, for essentially all first-order τ (i.e. no function types), the conclusions in this paper continue to hold; some extensions are given in Section 11. For arrays (or other large product types), the complexity is correct but constant-factor performance is rather lacking. *Efficient* support for arrays is left as future work.

https://doi.org/10.1017/S0956796825100051 Published online by Cambridge University Press

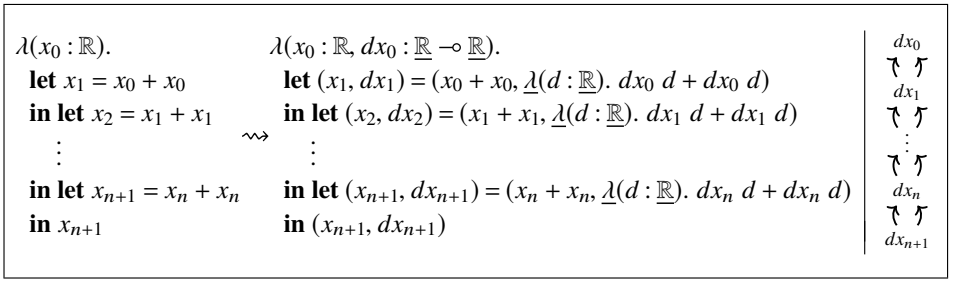


Fig. 2. Left: an example showing how naive dual-numbers reverse AD can result in exponential blow-up when applied to a program with sharing. Right: the dependency graph of the backpropagators dx_i.

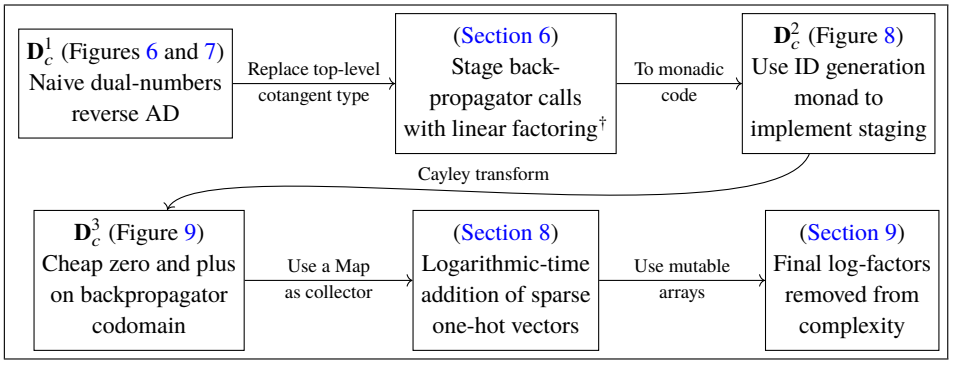


Fig. 3. Overview of the optimisations to dual-numbers reverse AD as a code transformation that are described in this paper. († = inspired by Brunel et al. (2020))

useless in practice—particularly because other (less flexible) reverse AD algorithms exist that indeed do a lot better (see e.g. Griewank and Walther (2008) and Baydin et al. (2017)).

Fortunately, 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 Figure 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. The last algorithm from Figure 3 can be enhanced to differentiate task-parallel source programs and can also be further optimised to something essentially equivalent to classical taping techniques.

Optimisation steps. The first step in Figure 3 is to 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 Figure 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

https://doi.org/10.1017/S0956796825100051 Published online by Cambridge University Press

able to contract applications of the same backpropagator using this linear factoring rule. The hope is that we can avoid executing f more often than is strictly necessary if we represent and reorganise these applications at runtime in a sufficiently clever way.

We achieve this linear factoring by not returning a plain c (presumably the type of (cotangents to) 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 $(\text{Staged}\ c, 0_{\text{Staged}}, (+_{\text{Staged}}))$, the only material change is that the calls to argument backpropagators in $\mathbf{D}^1_c[op]$ 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: at the very least, we need decidable equality on linear functions to be able to implement the linear factoring rule; and if we want any hope of an efficient implementation, we even need a linear order on such linear functions so that we can use them as keys in a tree map in the implementation of Staged. 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 (while this order turns out to be very important for efficiency).

We thus need two orders on our backpropagators. It turns out that for *sequential* input programs, it suffices to generate a unique, monotonically increasing identifier (ID) for each backpropagator that we create and use that ID not only as a witness for backpropagator identity and as a key in the tree map, but also as a witness for the dependency order (see below) of the backpropagators. These IDs are generated by letting the differentiated program run in an ID generation monad (a special case of a state monad). The result is shown in Figure 8, which is very similar to the previous version in Figure 6 apart from threading through the next-ID-to-generate. (On first glance, the code looks very different, but this is only due to monadic bookkeeping.)

At this point, the code transformation already 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 can be seen using the following observation: lambda functions in a pure functional program that do not take functions as arguments, can only call functions that appear in their closure. Because backpropagators are never mutually recursive (that could only happen if their corresponding scalars are defined mutually recursively, which, being scalars, would never terminate anyway), this observation means that calling backpropagators (really *unfolding*, as we delay evaluating subcalls) in their reverse dependency order achieves the goal of delaying the calls as long as possible. Thus, for monotonically increasing IDs, a backpropagator will only call other backpropagators with lower IDs. If we do not need parallelism in the differentiated programs, we can therefore suffice by simply resolving backpropagators from the highest to the lowest ID.

Effectively, one can think of the IDs as (labels of) the nodes in the computation graph of this run of the original program; from this perspective, the resolving process is nothing more than a choice of a reverse topological order of this graph, so that we can perform backpropagation (the reverse pass) in the correct order without redundant computation.

But we are not done yet. The code transformation at this point ($\mathbf{D}^2_c$ in Figure 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.

https://doi.org/10.1017/S0956796825100051 Published online by Cambridge University Press

This problem is somewhat more subtle because it is not actually apparent in the program transformation itself; indeed, looking back at Figure 1(c), there are no one-hot values to be found. However, the only way to *use* the program in Figure 1(c) 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. (These are created in the *wrapper* in Figure 7.) 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 7) and using a better sparse vector representation (Map $\mathbb{Z}\ \underline{\mathbb{R}}$ instead of a plain c value; Section 8). Cayley transformation (also known in the Haskell community by a common use: *difference lists* (Hughes, 1986)) is a classic technique in functional programming that represents an element m of a monoid M (written additively in this paper) 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 9). Such mutability can be safely accommodated in our code transformation by either swapping out the state monad for a resource-linear state monad, or by using mutable references in an `ST`-like monad (Section 9.1). (The latter can be generalised to the parallel case; see below.)

And then we are done for sequential programs, 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 at that input.

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 Lucatelli Nunes and Vákár (2024); 2. we transform this into our final algorithm using a combination of (A) standard optimisations that are well-known to preserve semantics (and hence correctness)—notably sparse vectors via Cayley transformation[9] and using a mutable array to optimise a tree map—and (B) the custom optimisation of linear factoring, which is semantics-preserving because derivatives (backpropagators) are linear functions.

Parallelism. If the user is satisfied with a fully sequential (non-parallel) computation of the derivative, it is enough to generate monotonically increasing integers and use them, together with their linear order, as IDs for backpropagators during the forward pass. This is the approach described so far.

[9] See Hughes (1986) for intuition, or Boisseau and Gibbons (2018, §3.3) for an explanation of the theory.

https://doi.org/10.1017/S0956796825100051 Published online by Cambridge University Press

However, if the source program has (task) parallelism (we assume fork-join parallelism in this paper), we would prefer to preserve that parallelism when performing backpropagation on the (implied) computation graph. The linear order (chronological, by comparing IDs) that we were using to witness the dependency order so far does not suffice any more: we need to be more frugal with adding spurious edges in the dependency graph that only exist because one backpropagator happened to be created after another on the clock. However, we only need to make our order (i.e. dependency graph) precise enough that independent tasks are incomparable in the order (and hence independent in the dependency graph); recording more accurate (lack of) dependencies between individual scalar operations would even allow exploiting implicit parallelism within an a priori serial subcomputation, which is potentially interesting but beyond the scope of this paper.[10]

Our solution is thus to switch from simple integer IDs to compound IDs, consisting of a job ID and a sequential ID within that job.[11] We assume parallelism in the source program is expressed using a parallel pair constructor with the following typing rule:

$$\frac{\Gamma \vdash t : \tau \qquad \Gamma \vdash s : \sigma}{\Gamma \vdash t \star s : (\tau, \sigma)}$$

(The method generalises readily to n-ary versions of this primitive.) To differentiate code using this construct, we take out the ID generation monad that the target program ran in so far, and replace it with a monad in which we can also record the dependency graph between parallel jobs. The derivative of $(\star)$ is the only place where we use the new methods of this monad: all other code transformation rules remain identical, save for writing the right-hand sides as a black-box monad instead of explicit state passing. We can then make use of this additional recorded information in the backpropagator resolution phase to do so in parallel; in this process, the net effect is that *forks from the primal become joins in the derivative computation*, and vice versa.

Comparison to other algorithms. We can relate the sequential version of our technique to that of Krawiec et al. (2022) by noting that we can replace $\mathbf{D}^1_c[\mathbb{R}] = (\mathbb{R}, \underline{\mathbb{R}} \multimap c)$ 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 10.

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 *runtime*, giving us the power to always achieve the desired reverse AD behaviour.

[10] It is also treacherous ground: it turns out to be very difficult to fulfil the promise of functional programming that all independent expressions are parallelisable, because thread management systems simply have too much overhead for that granularity of parallelism. (Interesting work here was done recently by Westrick et al. (2024).)

[11] Being pairs of integers, these still have a natural linear order to use as a map key.

https://doi.org/10.1017/S0956796825100051 Published online by Cambridge University Press

Finally, we are now in the position to note the similarity to (sequential[12]) taping-based AD as used by Kmett and Contributors (2021), older versions of PyTorch (Paszke et al., 2017), etc.: 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 (Section 9) which reifies this tape in a large array in the reverse pass, especially if one then proceeds to already use this array in the forward pass (Section 10.3), eliminates also this last difference.

3 Preliminaries: The complexity of reverse AD

The only reason, in practice, for using reverse AD over forward AD (which is significantly easier to implement) is computational complexity. Arguably, therefore, it is important that we fix precisely what the time complexity of reverse AD ought to be, and check that any proposed algorithm indeed conforms to this time complexity.

In this paper, we discuss a code transformation, so we phrase the desired time complexity in terms of a code transformation $\mathcal{R}$ that takes a program P of type $\mathbb{R}^n \to \mathbb{R}^m$ to a program $\mathcal{R}[P]$ of type[13] $(\mathbb{R}^n, \underline{\mathbb{R}}^m) \to \underline{\mathbb{R}}^n$ that computes the reverse derivative of P. The classic result (see Griewank and Walther, 2008) is that, for sequential first-order languages, $\mathcal{R}$ exists such that the following criterion is satisfied:

$$\exists c > 0.\ \forall P : \text{Programs}(\mathbb{R}^n \to \mathbb{R}^m).\ \forall I : \mathbb{R}^n, A : \underline{\mathbb{R}}^m.$$
$$\text{cost}(\mathcal{R}[P]\ (I, A)) \leq c \cdot (\text{cost}(P\ I) + \text{size}(I))$$

where we denote by $\text{cost}(E)$ the amount of time required to evaluate the expression E, and by $\text{size}(I)$ the amount of time required to read all of I sequentially. (Note that $\text{cost}(\mathcal{R}[P]\ (I, A))$ does not measure the cost of evaluating the code transformation $\mathcal{R}$ itself; that is considered to be a compile-time cost.) In particular, if P reads its entire input (and does not ignore part of it), the second line can be simplified to $\text{cost}(\mathcal{R}[P]\ (I, A))) \leq c \cdot \text{cost}(P\ I)$.

The most important point of this criterion is that c cannot depend on P: informally, the output program produced by reverse AD is not allowed to have more than a constant factor overhead in runtime over the original program, and this constant factor is uniform over all programs in the language.

A weaker form of the criterion is sometimes used where c is dependent on the program in question but not on the size of the input to that program; for example, '$f = \lambda arr.\ \text{sum}\ arr$' is allowed to have a different c than '$g = \lambda arr.\ \text{sum}\ (\text{map}\ (\lambda x.\ x + 1)\ arr)$', but given f, the same constant c will still apply regardless of the size of the *input array*. (Note that this can only make sense in a language that has variably sized arrays or similar structures.) This criterion is used by, e.g. Schenck et al. (2022), where c is proportional to the largest scope depth in the program. In this case, the criterion is expressed for a program family PF that

[12] Traditional taping-based reverse AD methods are fundamentally sequential. They may have parallel primitive operations, such as matrix multiplications, etc., but there is typically no general task parallelism.

[13] We will discuss the type of reverse AD in more detail in Section 4, generalised beyond vectors of reals.

https://doi.org/10.1017/S0956796825100051 Published online by Cambridge University Press

should be understood to be the same program for all n, just with different input sizes:

$$\forall PF : (n : \mathbb{N}) \to \text{Programs}(\mathbb{R}^n \to \mathbb{R}^m).\ \exists c > 0.\ \forall n \in \mathbb{N}.\ \forall I : \mathbb{R}^n, A : \underline{\mathbb{R}}^m.$$
$$\text{cost}(\mathcal{R}[PF_n]\ (I, A)) \leq c \cdot (\text{cost}(PF_n\ I) + \text{size}(I))$$

where c is preferably at most linearly or sub-linearly dependent on the size of the program code of PF.

The final sequential version of the code transformation described in this paper (in Section 9) satisfies the first (most stringent) criterion. For the parallel version (Section 12), the criterion holds for the amount of work performed.

4 Preliminaries: The type of reverse AD

Before one can define an algorithm, one has to fix the type of that algorithm. Similarly, before one can define a code transformation, one has to fix the domain and codomain of that transformation: the "type" of the transformation.

Typing forward AD. For forward AD on first-order programs (or at least, programs for which the input and output does not contain function values), the desired type seems quite evident: $\mathcal{F} : (\sigma \to \tau) \rightsquigarrow ((\sigma, \underline{\sigma}) \to (\tau, \underline{\tau}))$, where we write $T_1 \rightsquigarrow T_2$ for a (compiler) code transformation taking a program of type T_1 and returning a program of type T_2, and where $\underline{\tau}$ is the type of tangent vectors (derivatives) of values of type τ. This distinction between the type of values τ and the type of their derivatives $\underline{\tau}$ is important in some versions of AD, but will be mostly cosmetic in this paper; in an implementation one can take $\underline{\tau} = \tau$, but there is some freedom in this choice.[14] Given a program $f : \sigma \to \tau$, $\mathcal{F}[f]$ is a program that takes, in addition to its regular argument, also a tangent at that argument; the output is then the regular result paired up with corresponding tangent at that result.

More specifically, for forward AD, we want the following in the case that $\sigma = \mathbb{R}^n$ and $\tau = \mathbb{R}^m$ (writing $\mathbf{x} = (x_1, \ldots, x_n)$):[15]

$$\mathcal{F}[f]\left(\mathbf{x}, \left(\frac{\partial x_1}{\partial \alpha}, \ldots, \frac{\partial x_n}{\partial \alpha}\right)\right) = \left(f(\mathbf{x}), \left(\frac{\partial f(\mathbf{x})_1}{\partial \alpha}, \ldots, \frac{\partial f(\mathbf{x})_m}{\partial \alpha}\right)\right)$$

Setting $\alpha = x_i$ means passing $(0, \ldots, 1, \ldots, 0)$ as the argument of type $\underline{\sigma}$ and computing the partial derivative with respect to x_i of $f(\mathbf{x})$. In other words, $\text{snd}(\mathcal{F}[f](x, dx))$ is the directional derivative of f at x in the direction dx.

A first attempt at typing reverse AD. For reverse AD, the desired type is less evident. A first guess would be:

$$\mathcal{R}_1 : (\sigma \to \tau) \rightsquigarrow ((\sigma, \underline{\tau}) \to (\tau, \underline{\sigma}))$$

with this intended meaning for $\sigma = \mathbb{R}^n$ and $\tau = \mathbb{R}^m$ (again writing $\mathbf{x} = (x_1, \ldots, x_n)$):

$$\mathcal{R}_1[f]\left(\mathbf{x}, \left(\frac{\partial \omega}{\partial f(\mathbf{x})_1}, \ldots, \frac{\partial \omega}{\partial f(\mathbf{x})_m}\right)\right) = \left(f(\mathbf{x}), \left(\frac{\partial \omega}{\partial x_1}, \ldots, \frac{\partial \omega}{\partial x_n}\right)\right)$$

[14] For example, $\underline{\mathbb{R}} = \mathbb{R}$, but for $\mathbb{Z}$ one can choose the unit type () and be perfectly sound and consistent.

[15] This generalises to more complex (but still first-order) in/outputs by regarding those as collections of real values as well.

https://doi.org/10.1017/S0956796825100051 Published online by Cambridge University Press

In particular, if $\tau = \mathbb{R}$ and we pass 1 as its cotangent (also called adjoint) of type $\underline{\tau} = \underline{\mathbb{R}}$, the $\underline{\sigma}$-typed output contains the gradient with respect to the input.

Dependent types. However, $\mathcal{R}_1$ is not readily implementable for even moderately interesting languages. One way to see this is to acknowledge the reality that the type $\underline{\tau}$ (of derivatives of values of type τ) should really be dependent on the accompanying *primal value* of type τ. Let us write the type of derivatives at τ not as $\underline{\tau}$ but as $\mathcal{D}[\tau](x)$, where $x : \tau$ is that primal value. With just scalars and product types this dependence does not yet occur (e.g. $\mathcal{D}[\mathbb{R}](x) = \mathbb{R}$ independent of the primal value x), but when adding sum types (coproducts), the dependence becomes non-trivial: the only sensible derivatives for a value $\mathrm{inl}(x) : \sigma + \tau$ (for $x : \sigma$) are of type $\underline{\sigma}$. Letting $\underline{\sigma + \tau} = \underline{\sigma} + \underline{\tau}$ would allow passing a derivative value of type $\underline{\tau}$ to $\mathrm{inl}(x) : \sigma + \tau$, which is nonsensical (and an implementation could do little else than return a bogus value like 0 or throw a runtime error). The derivative of $\mathrm{inl}(2x) : \mathbb{R} + \mathrm{Bool}$ cannot be inr(True); it should at least somehow contain a real value.

Similarly, the derivative for a dynamically sized array, if the input language supports those, must really be of the same size as the input array. This, too, is a dependence of the type of the derivative on the *value* of the input.

Therefore, the output type of forward AD which we wrote above as $(\sigma, \underline{\sigma}) \to (\tau, \underline{\tau})$ should really[16] be $(\Sigma_{x:\sigma}\ \mathcal{D}[\sigma](x)) \to (\Sigma_{y:\tau}\ \mathcal{D}[\tau](y))$, rendering what were originally pairs of value and tangent now as *dependent* pairs of value and tangent. This is a perfectly sensible type, and indeed correct for forward AD, but it does not translate at all well to reverse AD in the form of $\mathcal{R}_1$: the output type would be something like $(\Sigma_{x:\sigma}\ \mathcal{D}[\tau](y)) \to (\Sigma_{y:\tau}\ \mathcal{D}[\sigma](x))$, which is nonsense because both x and y are out of scope.

Let-bindings. A different way to see that the type of $\mathcal{R}_1$ is unusable, is to note that one cannot even differentiate let-bindings using $\mathcal{R}_1$. In order to apply to (an extension of) the lambda calculus, let us rewrite the types somewhat: where we previously put a function $f : \sigma \to \tau$, we now put a term $x : \sigma \vdash t : \tau$ with its input in a free variable and producing its output as the returned value. Making the modest generalisation to support any full environment as input (instead of just a single variable), we get $\mathcal{R}_1 : (\Gamma \vdash t : \tau) \rightsquigarrow (\Gamma, d : \underline{\tau} \vdash \mathcal{R}_1[t] : (\tau, \underline{\Gamma}))$, where $\underline{\Gamma}$ is a tuple containing the derivatives of all elements in the environment Γ. (To be precise, we define $\underline{\varepsilon} = ()$ for the empty environment and $\underline{\Gamma, x : \tau} = (\underline{\Gamma}, \underline{\tau})$ inductively.)

Now, consider differentiating the following program using $\mathcal{R}_1$:

$$\Gamma \vdash (\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2) : \tau$$

where $\Gamma \vdash e_1 : \sigma$ and $\Gamma, x : \sigma \vdash e_2 : \tau$. Substituting, we see that $\mathcal{R}_1$ needs to somehow build a program of this type:

$$\Gamma, d : \underline{\tau} \vdash \mathcal{R}_1[\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2] : (\tau, \underline{\Gamma}) \tag{4.1}$$

However, recursively applying $\mathcal{R}_1$ on e_1 and e_2 yields terms:

$$\Gamma, d : \underline{\sigma} \vdash \mathcal{R}_1[e_1] : (\sigma, \underline{\Gamma})$$

[16] The notation '$\Sigma_{x:\sigma}\ \tau$' denotes a *sigma type*: it is roughly equivalent to the pair type (σ, τ), but the type τ is allowed to refer to x, the value of the first component of the pair.

https://doi.org/10.1017/S0956796825100051 Published online by Cambridge University Press

$$\Gamma, x : \sigma, d : \underline{\tau} \vdash \mathcal{R}_1[e_2] : (\tau, (\underline{\Gamma}, \underline{\sigma}))$$

To produce the program in Equation (4.1), we cannot use $\mathcal{R}_1[e_2]$ because we do not yet have an $x : \sigma$ (which needs to come from $\mathcal{R}_1[e_1]$), and we cannot use $\mathcal{R}_1[e_1]$ because the $\underline{\sigma}$ needs to come from $\mathcal{R}_1[e_2]$! The type of $\mathcal{R}_1$ demands the cotangent of the result *too early*.

Of course, one might argue that we can just use e_1 to compute the σ, $\mathcal{R}_1[e_2]$ to get the $\underline{\sigma}$ and e_2's contribution to $\underline{\Gamma}$, and finally $\mathcal{R}_1[e_1]$ to get e_1's contribution to $\underline{\Gamma}$ based on its own cotangent of type $\underline{\sigma}$. However, this would essentially compute e_1 twice (once directly and once as part of $\mathcal{R}_1[e_1]$), meaning that the time complexity becomes super-linear in the depth of let-bindings, which is quite disastrous for typical functional programs.

So in addition to not being precisely typeable, $\mathcal{R}_1$ is also not implementable in a compositional way.

Fixing the type of reverse AD. Both when looking at the dependent type of $\mathcal{R}_1$ and when looking at its implementation, we found that the cotangent $dy : \mathcal{D}[\tau](y)$ was required before the result $y : \tau$ was itself computed. One way to solve this issue is to just postpone requiring the cotangent of y, i.e. to instead look at $\mathcal{R}_2$:[17]

$$\mathcal{R}_2 : (\sigma \to \tau) \rightsquigarrow (\sigma \to (\tau, \underline{\tau} \to \underline{\sigma})) \qquad \text{(non-dependent version)}$$
$$\mathcal{R}_2 : (\sigma \to \tau) \rightsquigarrow (\Pi_{x:\sigma}\ \Sigma_{y:\tau}\ (\mathcal{D}[\tau](y) \to \mathcal{D}[\sigma](x))) \qquad \text{(dependent version)}$$

Note that this type *is* well-scoped. Furthermore, this "derivative function" mapping the cotangent of the result to the cotangent of the argument is actually a linear function, in the sense of a vector space homomorphism: indeed, it is multiplication by the Jacobian matrix of f, the input function. Thus we can write:

$$\mathcal{R}_2 : (\sigma \to \tau) \rightsquigarrow (\Pi_{x:\sigma}\ \Sigma_{y:\tau}\ (\mathcal{D}[\tau](y) \multimap \mathcal{D}[\sigma](x)))$$

which is the type of the reverse[18] AD code transformation derived by Elliott (2018) and in CHAD (e.g. Vákár and Smeding (2022); see also Section 15.3).[19]

While this formulation of reverse AD admits a rich mathematical foundation (Nunes and Vákár, 2023) and has the correct complexity (Smeding and Vákár, 2024), the required program transformation is more complex than the formulation $\mathcal{F}$ that we have for forward AD. In particular, we need to compute for each programming language construct what its CHAD transformation is, which may be non-trivial (for example, for the case of function types). This difficulty motivates us to pursue a reverse AD analogue of $\mathcal{F}$.

[17] Where a Σ-type is a dependent pair, a Π-type is a dependent function: $\Pi_{x:\sigma}\ \tau$ means $\sigma \to \tau$ except that the type τ may depend on the argument value x.

[18] There is also a corresponding formulation of forward AD which would have type:

$$\mathcal{F}_2 : (\sigma \to \tau) \rightsquigarrow (\Pi_{x:\sigma}\ \Sigma_{y:\tau}\ (\mathcal{D}[\sigma](x) \multimap \mathcal{D}[\tau](y)))$$

However, in the case of forward AD, there is no added value in using this more precise type, compared to our previous formulation $\mathcal{F}$. In fact, there are downsides: as we are forced to consume tangents only after the primal computation has finished, we can no longer interleave the primal and tangent computations, leading to larger memory use. Moreover, the resulting code transformation is more complex than $\mathcal{F}$.

[19] Actually, CHAD has non-identity type mappings for the primal types $x : \sigma$ and $y : \tau$ as well in order to compositionally support function values in a way that fits the type of $\mathcal{R}_2$. We consider only the top-level type in this discussion, and for first-order in- and output types, the two coincide.

https://doi.org/10.1017/S0956796825100051 Published online by Cambridge University Press

Applying Yoneda/CPS. An instance of the Yoneda lemma (or in this case: continuation-passing style; see also Boisseau and Gibbons (2018)) is that $\sigma \multimap \tau$ is equivalent to $\forall r.\, (\sigma \multimap r) \to (\tau \multimap r)$. We can apply this to the $\multimap$-arrow in $\mathcal{R}_2$ to obtain a type for reverse AD that is somewhat reminiscent of our formulation $\mathcal{F}$ of forward AD. With just Yoneda, we get $\mathcal{R}_3'''$ below; we then weaken this type somewhat by enlarging the scope of the $\forall c$ quantifier, weaken some more by taking the $(\mathcal{D}[\tau](x) \multimap c)$ argument before returning y, and finally we uncurry to arrive at $\mathcal{R}_3$:

$$\begin{aligned}
\mathcal{R}_3''' &: (\sigma \to \tau) \rightsquigarrow \Pi_{x:\sigma}\ \Sigma_{y:\tau}\ \forall c.\, ((\mathcal{D}[\sigma](x) \multimap c) \to (\mathcal{D}[\tau](y) \multimap c)) \\
\mathcal{R}_3'' &: (\sigma \to \tau) \rightsquigarrow \forall c.\, \Pi_{x:\sigma}\ \Sigma_{y:\tau}\ ((\mathcal{D}[\sigma](x) \multimap c) \to (\mathcal{D}[\tau](y) \multimap c)) \\
\mathcal{R}_3' &: (\sigma \to \tau) \rightsquigarrow \forall c.\, \Pi_{x:\sigma}\ ((\mathcal{D}[\sigma](x) \multimap c) \to \Sigma_{y:\tau}\ (\mathcal{D}[\tau](y) \multimap c)) \\
\mathcal{R}_3 &: (\sigma \to \tau) \rightsquigarrow \forall c.\, (\Sigma_{x:\sigma}\ (\mathcal{D}[\sigma](x) \multimap c)) \to \Sigma_{y:\tau}\ (\mathcal{D}[\tau](y) \multimap c) \\
&\qquad\qquad \forall c.\, (\sigma, \quad \underline{\sigma} \qquad\quad \multimap c\ ) \to (\tau, \quad \underline{\tau} \qquad\quad \multimap c)
\end{aligned}$$

We give both a dependently typed (black) and a simply typed (grey) signature for $\mathcal{R}_3$.

The $\multimap$-arrows in these types, as well as the c bound by the $\forall$-quantifier, live in the category of commutative monoids. Indeed, c will always have a commutative monoid structure in this paper; that is: it has a zero $\underline{0}$ as well as a commutative, associative addition operation $(+) : (c, c) \to c$ for which $\underline{0}$ is the unit. (The $\multimap$-arrows in these types are really vector space homomorphisms, but since we will only use the substructure of commutative monoids in this paper, and forget about scalar multiplication, we will always consider $\multimap$-functions (commutative) monoid homomorphisms.)

Returning to the types in question, we see that we can convert $\mathcal{R}_2[t]$ to $\mathcal{R}_3[t]$:

$$\begin{aligned}
&(\lambda(x : \sigma, dx : \mathcal{D}[\sigma](x) \multimap c). \\
&\qquad \textbf{let}\ (y : \tau, dy : \mathcal{D}[\tau](y) \multimap \mathcal{D}[\sigma](x)) = \mathcal{R}_2[t]\ x\ \textbf{in}\ (y, dx \circ dy)) \\
&\quad : \forall c.\, (\Sigma_{x:\sigma}\ (\mathcal{D}[\sigma](x) \multimap c)) \to \Sigma_{y:\tau}\ (\mathcal{D}[\tau](y) \multimap c)
\end{aligned}$$

where we write $\circ$ for the composition of linear functions. We can also convert $\mathcal{R}_3[t]$ back to $\mathcal{R}_2[t]$, but due to how we weakened the types above, only in the non-dependent world:

$$(\lambda(x : \sigma).\, \mathcal{R}_3[t]\ (x, \underline{\lambda}(z : \underline{\sigma}).\, z)) : \sigma \to (\tau, \underline{\tau} \multimap \underline{\sigma})$$

So, in some sense, $\mathcal{R}_2$ and $\mathcal{R}_3$ compute the same thing, albeit with types that differ in how precisely they portray the dependencies.

In fact, $\mathcal{R}_3$ admits a very elegant implementation as a program transformation that is structurally recursive over all language elements except for the primitive operations in the leaves. However, there are some issues with the computational complexity of this straightforward implementation of $\mathcal{R}_3$, one of which we will fix here immediately, and the other of which are the topic of the rest of this paper.

Moving the pair to the leaves. Let us return to forward AD for a moment. Recall the type we gave for forward AD:[20]

$$\mathcal{F} : (\sigma \to \tau) \rightsquigarrow ((\sigma, \underline{\sigma}) \to (\tau, \underline{\tau}))$$

[20] We revert to the non-dependent version for now because the dependencies are irrelevant for this point, and they clutter the presentation.

https://doi.org/10.1017/S0956796825100051 Published online by Cambridge University Press

Supposing we have a program $f : ((\mathbb{R}_1, \mathbb{R}_2), \mathbb{R}_3) \to \mathbb{R}_4$, we get: (the subscripts are semantically meaningless and are just for tracking arguments)

$$\mathcal{F}[\![f]\!] : (((\mathbb{R}_1, \mathbb{R}_2), \mathbb{R}_3), ((\underline{\mathbb{R}_1}, \underline{\mathbb{R}_2}), \underline{\mathbb{R}_3})) \to (\mathbb{R}_4, \underline{\mathbb{R}_4})$$

While this is perfectly implementable and correct and efficient, it is not the type that corresponds to what is by far the most popular implementation of forward AD, namely *dual-numbers forward AD*, which has the following type:

$$\mathcal{F}_{\text{dual}} : (\sigma \to \tau) \rightsquigarrow (\text{Dual}[\sigma] \to \text{Dual}[\tau])$$
$$\text{Dual}[\mathbb{R}] = (\mathbb{R}, \underline{\mathbb{R}}) \qquad \text{Dual}[()] = () \qquad \text{Dual}[(\sigma, \tau)] = (\text{Dual}[\sigma], \text{Dual}[\tau])$$

Intuitively, instead of putting the pair at the root like $\mathcal{F}$ does, $\mathcal{F}_{\text{dual}}$ puts the pair at the leaves—more specifically, at the scalars in the leaves, leaving non-$\mathbb{R}$ types like () or $\mathbb{Z}$ alone. For the given example program f, dual-numbers forward AD would yield the following derivative program type:

$$\mathcal{F}_{\text{dual}}[\![f]\!] : (((\mathbb{R}_1, \underline{\mathbb{R}_1}), (\mathbb{R}_2, \underline{\mathbb{R}_2})), (\mathbb{R}_3, \underline{\mathbb{R}_3})) \to (\mathbb{R}_4, \underline{\mathbb{R}_4})$$

Of course, for any given types σ, τ the two versions are trivially inter-converted, and as stated, for forward AD both versions can be defined inductively equally well, resulting in efficient programs in terms of time complexity.

However, for reverse AD in the style of $\mathcal{R}_3$, the difference between $\mathcal{R}_3$ and its pair-at-the-leaves dual-numbers variant ($\mathcal{R}_{3\text{dual}}$ below) is more pronounced. First note that indeed both styles (with the pair at the root and with the pair at the leaves) produce a sensible type for reverse AD: (again for $f : ((\mathbb{R}_1, \mathbb{R}_2), \mathbb{R}_3) \to \mathbb{R}_4$)

$$\mathcal{R}_3[\![f]\!] : \forall c.\ (((\mathbb{R}_1, \mathbb{R}_2), \mathbb{R}_3), ((\underline{\mathbb{R}_1}, \underline{\mathbb{R}_2}), \underline{\mathbb{R}_3}) \multimap c) \to (\mathbb{R}_4, \underline{\mathbb{R}_4} \multimap c)$$
$$\mathcal{R}_{3\text{dual}}[\![f]\!] : \forall c.\ (((\mathbb{R}_1, \underline{\mathbb{R}_1} \multimap c), (\mathbb{R}_2, \underline{\mathbb{R}_2} \multimap c)), (\mathbb{R}_3, \underline{\mathbb{R}_3} \multimap c)) \to (\mathbb{R}_4, \underline{\mathbb{R}_4} \multimap c)$$

The individual functions of type $\underline{\mathbb{R}} \multimap c$ are usually called *backpropagators* in literature, and we will adopt this terminology.

Indeed, these two programs are again easily inter-convertible, if one realises that:

1. c is a commutative monoid and thus possesses an addition operation, which can be used to combine the three c results into one for producing the input of $\mathcal{R}_3$ from the input of $\mathcal{R}_{3\text{dual}}$;
2. The function $g : ((\underline{\mathbb{R}_1}, \underline{\mathbb{R}_2}), \underline{\mathbb{R}_3}) \multimap c$ is linear, and hence, e.g. $\lambda(x : \underline{\mathbb{R}_2}).\ g\ ((0, x), 0)$ suffices as value for $\underline{\mathbb{R}_2} \multimap c$.

However, the problem arises when defining $\mathcal{R}_3$ inductively as a program transformation. To observe this difference between $\mathcal{R}_3$ and $\mathcal{R}_{3\text{dual}}$, consider the term $t = \lambda(x : (\sigma, \tau)).\ \text{fst}(x)$ of type $(\sigma, \tau) \to \sigma$ and the types of its derivative using both methods:

$$\mathcal{R}_3[\![t]\!] : \forall c.\ ((\sigma, \tau), (\underline{\sigma}, \underline{\tau}) \multimap c) \to (\sigma, \underline{\sigma} \multimap c)$$
$$\mathcal{R}_{3\text{dual}}[\![t]\!] : \forall c.\ (\text{Dual}_c[\sigma], \text{Dual}_c[\tau]) \to \text{Dual}_c[\sigma]$$
$$\text{Dual}_c[\mathbb{R}] = (\mathbb{R}, \underline{\mathbb{R}} \multimap c) \qquad \text{Dual}_c[()] = () \qquad \text{Dual}_c[(\sigma, \tau)] = (\text{Dual}_c[\sigma], \text{Dual}_c[\tau])$$

Their implementations look as follows:

$$\mathcal{R}_3[\![t]\!] = \lambda(x : (\sigma, \tau), dx : (\underline{\sigma}, \underline{\tau}) \multimap c).\ (\text{fst}(x), \lambda(d : \underline{\sigma}).\ dx\ (d, \underline{0}_{\underline{\tau}}))$$
$$\mathcal{R}_{3\text{dual}}[\![t]\!] = \lambda(x : (\text{Dual}_c[\sigma], \text{Dual}_c[\tau])).\ \text{fst}(x)$$

https://doi.org/10.1017/S0956796825100051 Published online by Cambridge University Press

where $\underline{0}_{\underline{\tau}}$ is the zero value of the cotangent type of τ. The issue with the first variant is that τ may be an arbitrarily complex type, perhaps even containing large arrays of scalars, and hence this zero value $\underline{0}_{\underline{\tau}}$ may also be large. Having to construct this large zero value is not, in general, possible in constant time, whereas the primal operation (fst) was a constant-time operation; this is anathema to getting a reverse AD code transformation with the correct time complexity. Further, on our example program, we see that the variant $\mathcal{R}_3$ results in a more complex code transformation than $\mathcal{R}_{3\text{dual}}$, and this observation turns out to hold more generally. $\mathcal{R}_3$ shares both these challenges with the CHAD formulation $\mathcal{R}_2$ of reverse AD.

As evidenced by the complexity analysis and optimisation of the CHAD reverse AD algorithm (Smeding and Vákár, 2024), there are ways to avoid having to construct a non-constant-size zero value here. In fact, we use one of those ways, in a different guise, later in this paper in Section 7. However, in this paper, we choose the $\mathcal{R}_{3\text{dual}}$ approach. We pursue the dual numbers approach not to avoid having to deal with the issue of large zeros—indeed, skipping the problem here just moves it somewhere else, namely to the implementation of the backpropagators ($\underline{\mathbb{R}} \multimap c$). Rather, we pursue this approach because $\mathcal{R}_{3\text{dual}}$ extends more easily to a variety of language features (see Sections 11 and 12).

5 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. In subsequent sections, we will apply a number of optimisations to this algorithm that fix the complexity issues, to derive an algorithm that does have the desired complexity.

5.1 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 Figure 4; the initial target language is given in Figure 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 n-ary primitive operations op on real numbers in the source language. Concretely, given typed programs $\Gamma \vdash t_i : \mathbb{R}$ of type $\mathbb{R}$ in typing context Γ, for $1 \leq i \leq n$, we have a program $\Gamma \vdash op(t_1, \ldots, t_n) : \mathbb{R}$. The program transformation does not care what the contents of Op_n are, as long as the partial derivatives are available in the target language after differentiation.

In the target language in Figure 5, we add linear functions with the type $\sigma \multimap \tau$: these functions are linear in the sense of being monoid homomorphisms, meaning that $f(0) = 0$ and $f(x + y) = f(x) + f(y)$ if $f : \sigma \multimap \tau$. Because it is not well defined what the derivative of

https://doi.org/10.1017/S0956796825100051 Published online by Cambridge University Press

Types:

$$\sigma, \tau \ ::= \ \mathbb{R} \mid () \mid (\sigma, \tau) \mid \sigma \to \tau \mid \mathbb{Z}$$

Terms:

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

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

Types:

$$\begin{array}{rcll} \overline{\sigma}, \overline{\tau} & ::= & \mathbb{R} \mid () \mid (\overline{\sigma}, \overline{\tau}) \mid \mathbb{Z} & \text{(types without functions)} \\ \sigma, \tau & ::= & \mathbb{R} \mid () \mid (\sigma, \tau) \mid \mathbb{Z} \mid \sigma \to \tau & \\ & \mid & \overline{\sigma} \multimap \overline{\tau} & \text{(linear functions)} \end{array}$$

Terms:

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

Linear function bodies:

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

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

a function value (in the input or output of a program) should be, we disallow function types on either side of the —∘-arrow.[21] (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 —∘-arrows are replaced with → (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

[21] In Section 7, we will, actually, put endomorphisms ($a \to a$) on both sides of a —∘-arrow; for justification, see there.

https://doi.org/10.1017/S0956796825100051 Published online by Cambridge University Press

On types:

$$\mathbf{D}^1_c[\mathbb{R}] = (\mathbb{R}, \underline{\mathbb{R}} \multimap c) \qquad \mathbf{D}^1_c[()] = () \qquad \mathbf{D}^1_c[(\sigma, \tau)] = (\mathbf{D}^1_c[\sigma], \mathbf{D}^1_c[\tau])$$
$$\mathbf{D}^1_c[\sigma \to \tau] = \mathbf{D}^1_c[\sigma] \to \mathbf{D}^1_c[\tau] \qquad \mathbf{D}^1_c[\mathbb{Z}] = \mathbb{Z}$$

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

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

$\mathbf{D}^1_c[\mathbf{let}\ x : \tau = s\ \mathbf{in}\ t] =$
$\quad \mathbf{let}\ x : \mathbf{D}^1_c[\tau] = \mathbf{D}^1_c[s]\ \mathbf{in}\ \mathbf{D}^1_c[t]$

$\mathbf{D}^1_c[()] = ()$

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

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

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

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

$\mathbf{D}^1_c[s\ t] = \mathbf{D}^1_c[s]\ \mathbf{D}^1_c[t]$

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

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

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

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 need to compute the linear (reverse) derivatives of the primitive operations. For every $op \in \mathrm{Op}_n$, we require chosen programs $\Gamma \vdash \partial_i op(t_1, \ldots, t_n) : \underline{\mathbb{R}} \multimap \underline{\mathbb{R}}$, given $\Gamma \vdash t_i : \mathbb{R}$, for $1 \leq i \leq n$. We require that these implement the partial derivatives of op in the sense that they have semantics $\partial_i op(x)(d) = d \cdot \frac{\partial(op(x))}{\partial x_i}$.

5.2 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 Figure 6. 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.

https://doi.org/10.1017/S0956796825100051 Published online by Cambridge University Press

The transformation consists of a mapping $\mathbf{D}_c^1[\tau]$ on types τ and a mapping $\mathbf{D}_c^1[t]$ on terms t.[22] 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, 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*. If a $\mathbf{D}_c^1$-transformed program at some point computes a scalar–backpropagator pair (x, d) from a top-level input $input : \sigma$, then given a $z : \underline{\mathbb{R}}$, $d(z) : \underline{\sigma}$ is equal to z times the gradient of x as a function of $input$.

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 abstractions and let-bindings, the types of the variables do change.

Scalar constants are transformed to a pair of that scalar constant and a backpropagator for that constant. Because a constant clearly does not depend on the input at all, its gradient is zero, and hence the backpropagator is identically zero, thus $\underline{\lambda}(z : \underline{\mathbb{R}}).\ \underline{0}$.

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. Now, writing α for the top-level program input, we have:

$$z \cdot \frac{\partial(op(x_1,\ldots,x_n))}{\partial\alpha} = z \cdot \sum_{i=1}^{n} \frac{\partial(op(x_1,\ldots,x_n))}{\partial x_i} \cdot \frac{\partial x_i}{\partial\alpha} = \sum_{i=1}^{n} \frac{\partial x_i}{\partial\alpha} \cdot \left(z \cdot \frac{\partial(op(x_1,\ldots,x_n))}{\partial x_i}\right)$$

and because $d_i\ z = \frac{\partial x_i}{\partial\alpha}$ and $\partial_i op(x_1, \ldots, x_n) = \frac{\partial(op(x_1,\ldots,x_n))}{\partial x_i}$, the appropriate backpropagator to return is indeed $\underline{\lambda}(z : \underline{\mathbb{R}}).\ \sum_{i=1}^{n} d_i\ (\partial_i op(x_1, \ldots, x_n))$ as is written in Figure 6. This sum is on values of type c, which is currently still the type of the top-level program input.

Wrapper of the AD transformation. We want the external API of the AD transformation to be like $\mathcal{R}_2$ from Section 4:

$$\mathcal{R}_2[f] : \sigma \to (\tau, \underline{\tau} \to \underline{\sigma})$$

given $f : \sigma \to \tau$. However, our compositional code transformation actually follows $R_{3\text{dual}}$:

$$\mathcal{R}_{3\text{dual}}[t] : \forall c.\ \mathbf{D}_c^1[\sigma] \to \mathbf{D}_c^1[\tau]$$

hence we need to convert from $\mathcal{R}_{3\text{dual}}$ form to the intermediate $\mathcal{R}_3$:

$$\mathcal{R}_3[t] : \forall c.\ (\sigma, \underline{\sigma} \multimap c) \to (\tau, \underline{\tau} \multimap c)$$

and from there to $\mathcal{R}_2$. The conversion from $(\sigma, \sigma \multimap c)$ to $\mathbf{D}_c^1[\sigma]$, for first-order σ, consists of *interleaving* the backpropagator into the data structure of type σ; the converse (for τ) is a similar deinterleaving process. These two conversions (back and forth) are implemented by Interleave^1 and Deinterleave^1 in Figure 7. The final conversion from $\mathcal{R}_3$ to $\mathcal{R}_2$ is easy in

[22] In this section we choose c to be the domain type of the top-level program; later we will modify c to support our optimisations.

https://doi.org/10.1017/S0956796825100051 Published online by Cambridge University Press

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

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

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

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

the simply-typed world (as described in Section 4); this conversion is implemented in the top-level wrapper, Wrap^1, also in Figure 7.

5.3 Running example

Let us look at the simple example from Figure 1(a) in Section 2:

$$\lambda(x : \mathbb{R}, y : \mathbb{R}).\ \underbrace{\mathbf{let}\ z = x + y\ \mathbf{in}\ x \cdot z}_{t} \tag{5.1}$$

We have $x : \mathbb{R}, y : \mathbb{R} \vdash t : \mathbb{R}$. The code transformation $\mathbf{D}^1_c$ from Figure 6 maps t to:

$$
\begin{aligned}
\mathbf{D}^1_c[t] = \mathbf{let}\ z = &\ \mathbf{let}\ (x_1, d_1) = x\ \mathbf{in\ let}\ (x_2, d_2) = y \\
&\ \mathbf{in}\ (x_1 + x_2, \underline{\lambda}(z' : \underline{\mathbb{R}}).\ d_1\ z' + d_2\ z') \\
\mathbf{in\ let}\ (x_1, d_1) = &\ x\ \mathbf{in\ let}\ (x_2, d_2) = z \\
&\ \mathbf{in}\ (x_1 \cdot x_2, \underline{\lambda}(z' : \underline{\mathbb{R}}).\ d_1\ (z \cdot z') + d_2\ (x \cdot z'))
\end{aligned}
$$

which satisfies $x : (\mathbb{R}, \underline{\mathbb{R}} \multimap c), y : (\mathbb{R}, \underline{\mathbb{R}} \multimap c) \vdash \mathbf{D}^1_c[t] : (\mathbb{R}, \underline{\mathbb{R}} \multimap c)$. (We α-renamed z from Figure 6 to z' here.) The wrapper Wrap^1 in Figure 7 computes, given $x : (\mathbb{R}, \mathbb{R})$:

$$\mathrm{Interleave}^1_{(\mathbb{R},\mathbb{R})}\ (x, \underline{\lambda}(z : (\underline{\mathbb{R}}, \underline{\mathbb{R}})).\ z) = ((\mathrm{fst}(x), \underline{\lambda}(z : \underline{\mathbb{R}}).\ (z, \underline{0})), (\mathrm{snd}(x), \underline{\lambda}(z : \underline{\mathbb{R}}).\ (\underline{0}, z)))$$

The x and y in Eq. (5.1) get bound to the first half and the second half of this pair, respectively. $\mathrm{Deinterleave}^1_\tau$ is the identity in this case, because $\tau = \mathbb{R}$.

In Sections 6.1 and 7.2, we will revisit this example to show how the outputs change.

https://doi.org/10.1017/S0956796825100051 Published online by Cambridge University Press

5.4 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; Lucatelli Nunes and Vákár, 2024). However, as given here, it does not at all have the right time complexity.

The forward pass is fine: calling the function $\mathrm{Wrap}^1[\lambda(x:\sigma).\ t:\tau]:\sigma\to(\tau,\tau\multimap\sigma)$ at some input $x:\sigma$ 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. Figure 2 in Section 2 displays an example program t with its naive derivative $\mathbf{D}^1_c[t]$, in which sharing of scalars thus 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 from a theoretical point of view by observing that calling a linear backpropagator multiple times is 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,[23] 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 define a custom operational semantics that symbolically evaluates the output program of the transformation to a straight-line program with the input backpropagators still as symbolic variables, and afterwards symbolically reduces 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 achieve linear factoring in a standard, call-by-value semantics for the target language. In doing so, we attain 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

[23] Obstacles to this are, e.g. $\underline{0}$ and $(+)$ on the type c; we will fix this in Sections 7, 8 and 9.

https://doi.org/10.1017/S0956796825100051 Published online by Cambridge University Press

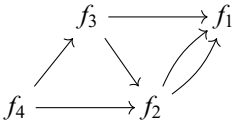
cotangents of the input that we have considered so far. Avoiding the need for a custom operational semantics allows the wrapper of our code transformation to be relatively small (though it will grow in subsequent sections), and the core of the differentiated program to run natively in the target language.

6 Linear factoring by staging function calls

As observed above in Section 5.4, 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 linear functions f satisfy $f\ x + f\ y = f\ (x + y)$. We must find a way to "merge" all invocations of a single backpropagator with this linear factoring rule so that in the end only one invocation remains (or zero if it was never invoked 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 (representative) situation where a program generates the following backpropagators:

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



Suppose 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 call graph on the right.

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.[24] Note that this uniform representation in a dictionary works because all backpropagators in the core transformed program (outside of the wrapper) have the same domain ($\underline{\mathbb{R}}$) and codomain (c). The idea is

[24] This is the intuition; it will not go through precisely as planned, but something similar will.

https://doi.org/10.1017/S0956796825100051 Published online by Cambridge University Press

that we replace what are now applications of linear functions with the creation of a dictionary containing one key-value (function-argument) pair 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 change $\mathbf{D}^1_c[\mathbb{R}] = (\mathbb{R}, \underline{\mathbb{R}} \multimap c)$ to instead read $\mathbf{D}^1_c[\mathbb{R}] = (\mathbb{R}, \underline{\mathbb{R}} \multimap \text{Staged}\, c)$, where 'Staged c' is our "dictionary".[25] We define, or rather, would like to define Staged c as follows: ('Map k v' is the usual type of persistent tree-maps with keys of type k and values of type v)

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

Suspending disbelief about implementability, 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 $(x, \{f_1 \mapsto a_1, f_2 \mapsto a_2\})$ of type 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 $\mathbf{D}^1_c$, 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 of 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:

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

- 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 $f : \underline{\mathbb{R}} \multimap c$ to an argument $x : \underline{\mathbb{R}}$ now gets replaced with SCall $f\ x \coloneqq (\underline{0}, \{f \mapsto x\}) : \text{Staged}\, c$. This occurs in $\mathbf{D}^1_c[op(...)]$ and in Deinterleave^1.

Essentially, this step of replacing c with Staged c can be seen as a clever partial defunctionalisation of our backpropagators.

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:

$$\text{SResolve} : (\text{Staged}\, c) \to c$$

will need to call functions stored 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

[25] In the wrapper, we still instantiate c to the domain type σ, meaning that Staged σ will actually appear in the derivative program.

https://doi.org/10.1017/S0956796825100051 Published online by Cambridge University Press

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 $\mathbf{D}^1_c$ from Figure 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 Figure 7, all backpropagators in the derivative program have one of the following three forms:

1. $(\underline{\lambda}(z : \underline{\mathbb{R}}).\ t)$ 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, after trivial beta-reduction of the intermediate linear functions, from the way that $\mathrm{Interleave}^1_\sigma$ (Figure 7) handles references to the global inputs of the program.
2. $(\underline{\lambda}(z : \underline{\mathbb{R}}).\ \underline{0})$ 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. $(\underline{\lambda}(z : \underline{\mathbb{R}}).\ d_1\ (\partial_1 op(x_1, \ldots, x_n)(z)) + \cdots + d_n\ (\partial_n op(x_1, \ldots, x_n)(z)))$ 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 *a backpropagator f_r paired with a scalar r will only ever call backpropagators f_s that are paired with scalars s, such that r already has a dependency on s in the source program*. In particular, f_s must have been created (at runtime of the derivative program) before f_r itself was created. Furthermore, f_r is not the same function as f_s because that would mean that r depends on itself in the source program. Therefore, if, at runtime, we define a partial order on backpropagators with the property that $f_r \geq f_s$ if r depends on s (and $f_r > f_s$ if they are not syntactically equal), we obtain that a called backpropagator is always strictly *lower* in the order than the backpropagator it was called from.

In practice, we achieve this by giving unique IDs, of some form, to backpropagators and defining a partial order on those IDs at runtime, effectively building a computation graph. This partial order tells us in which order to resolve backpropagators: we walk the order from top to bottom, starting from the maximal IDs and repeatedly resolving the predecessors in the order after we finish resolving a particular backpropagator. After all,

https://doi.org/10.1017/S0956796825100051 Published online by Cambridge University Press

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 use the ID as the Map key, which is perfectly valid and efficient as long as the IDs are chosen to be of some type that can be linearly ordered to perform binary search (such as tuples of integers).

We have still been rather vague about how precisely to assign the IDs and define their partial order. In fact, there is some freedom in how to do that. For the time being, we will simply work with *sequentially incrementing integer IDs with their linear order*, which suffices for sequential programs. Concretely, we number backpropagators with incrementing integer IDs at runtime, at the time of their creation by a $\underline{\lambda}$. We then resolve them from top to bottom, starting from the unique maximal ID. To support parallelism in Section 12, we will revisit this choice and work instead with *pairs* of integers (a combination of a job ID and a sequentially increasing ID within that job) with a partial order that encodes the fork-join parallelism structure of the source program. That choice of non-linear partial order allows us to reflect the parallelism present in the source program in a parallel reverse pass to compute derivatives. But because we can mostly separate the concerns of ID representation and differentiation, we will focus on simple, sequential integer IDs for now.

When we give backpropagators integer IDs, we can rewrite Staged c and SCall:

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

We call the second component of a Staged c value, which has type Map $\mathbb{Z}$ $(\underline{\mathbb{R}} \multimap$ Staged $c, \underline{\mathbb{R}})$, 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 $\mathbb{Z}$ to keep track of our integer IDs). The resulting new program transformation, modified from Figures 6 and 7, is shown in Figure 8.

New program transformation. In Figure 8, the term transformation now produces a term in the ID generation monad $(\mathbb{Z} \to (-, \mathbb{Z}))$; 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 $\mathbf{D}_c^2[\mathbb{R}]$, which now tags backpropagators with an ID): $\mathbf{D}_c^2[\sigma \to \tau]$ now produces a monadic function type instead of a plain function type.

On the term level, notice that the backpropagator for primitive operations (in $\mathbf{D}_c^2[op(...)]$) 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 $(+_{\text{Staged}})$ refers to map union including linear factoring; for example:

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

https://doi.org/10.1017/S0956796825100051 Published online by Cambridge University Press

On types:

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

On terms:

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

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

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

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

etc.

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

$\mathbf{D}^2_c[op(t_1, \ldots, t_n)] =$
$\lambda i.\ \textbf{let}\ ((x_1, d_1), i_1) = \mathbf{D}^2_c[t_1]\ i\ \textbf{in}\ \ldots\ \textbf{in let}\ ((x_n, d_n), i_n) = \mathbf{D}^2_c[t_n]\ i_{n-1}$
$\textbf{in}\ ((op(x_1, \ldots, x_n), (i_n, \underline{\lambda}(z : \underline{\mathbb{R}}).\ \text{SCall}\ d_1\ (\partial_1 op(x_1, \ldots, x_n)(z)) +_{\text{Staged}} \cdots$
$+_{\text{Staged}}\ \text{SCall}\ d_n\ (\partial_n op(x_1, \ldots, x_n)(z)))), i_n + 1)$

Changed wrapper:

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

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

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

Staged interface:

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

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

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

Fig. 8. The monadically transformed code transformation (from Figures 4 to 5 plus Staged operations), based on Figure 6. Grey parts are unchanged or simply monadically lifted.

https://doi.org/10.1017/S0956796825100051 Published online by Cambridge University Press

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. This invariant is quite essential in the algorithms in this paper.

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 $(\underline{\lambda}(z : \underline{\sigma}).\ z) : \underline{\sigma} \multimap \underline{\sigma}$ in Figure 7, has now become $\mathrm{SCotan} : \underline{\sigma} \multimap \mathrm{Staged}\ \underline{\sigma}$, which injects a cotangent into a Staged c object. Interleave^2 "splits" this function up into individual $\underline{\mathbb{R}} \multimap \mathrm{Staged}\ \underline{\sigma}$ 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 order from the highest ID to the lowest ID, we ensure that every backpropagator is called at most once.[26] This is done in an additional function, SResolve, which can be written as follows:

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

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

6.1 Running example

In Section 5.3, we looked at the term $x : \mathbb{R}, y : \mathbb{R} \vdash t = \textbf{let } z = x + y \textbf{ in } x \cdot z : \mathbb{R}$ from Figure 1(a). With the updated transformation from Figure 8, we now get (with lambda-application redexes already simplified for readability):

$$
\begin{aligned}
\mathbf{D}^2_c[t] = \lambda i_1.\ &\textbf{let } (z, i_4) = \textbf{let } ((x_1, d_1), i_2) = (x, i_1) \textbf{ in let } ((x_2, d_2), i_3) = (y, i_2)\\
&\qquad \textbf{in } ((x_1 + x_2, (i_3, \underline{\lambda}(z' : \underline{\mathbb{R}}).\ \mathrm{SCall}\ d_1\ z' +_{\mathrm{Staged}} \mathrm{SCall}\ d_2\ z')), i_3 + 1)\\
&\textbf{in let } ((x_1, d_1), i_5) = (x, i_4) \textbf{ in let } ((x_2, d_2), i_6) = (z, i_5)\\
&\quad \textbf{in } ((x_1 \cdot x_2, (i_6, \underline{\lambda}(z' : \underline{\mathbb{R}}).\ \mathrm{SCall}\ d_1\ (z \cdot z') +_{\mathrm{Staged}} \mathrm{SCall}\ d_2\ (x \cdot z'))), i_6 + 1)
\end{aligned}
$$

The result of $\mathrm{Interleave}^2_{(\mathbb{R},\mathbb{R})}\ (x, \mathrm{SCotan})\ 0$, as called from Wrap^2 in Figure 8, is:

$$(((\mathrm{fst}(x), (0, d_{\mathrm{in},0})), (\mathrm{snd}(x), (1, d_{\mathrm{in},1}))), 2)$$

where we abbreviated the input backpropagators as $d_{\mathrm{in},0} = \underline{\lambda}(z : \underline{\mathbb{R}}).\ ((z, \underline{0}), \{\})$ and $d_{\mathrm{in},1} = \underline{\lambda}(z : \underline{\mathbb{R}}).\ ((\underline{0}, z), \{\})$. Now, assuming that the input x is, say, $(12, 13)$ and that the initial cotangent is 1, the Staged $(\mathbb{R}, \mathbb{R})$ object that gets passed to SResolve in Wrap^2 (i.e. the

[26] Technically, some backpropagators (namely, the ones that appear in the top-level function output), are invoked more than once because Deinterleave^2 indiscriminately calls all output backpropagators. If the function output contains n scalars, this can lead to $O(n)$ overhead. (In particular, for a function $f : \tau \to \mathbb{R}$, there is no such overhead.) The complexity of the algorithm is not in fact compromised, because the size of the output is at most the runtime of the original function. If desired, Deinterleave^2 can be modified to return a Staged c object that *stages* calls to the output backpropagators, instead of directly calling them. We did not make this change for simplicity of presentation.

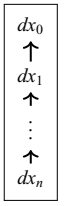
https://doi.org/10.1017/S0956796825100051 Published online by Cambridge University Press

result of $d\ z = d\ 1$ there) looks as follows:

$$(\underline{0} + \underline{0}, \{0 \mapsto (d_{\text{in},0}, 25) \\ ,2 \mapsto (\underline{\lambda}(z' : \underline{\mathbb{R}}).\ (\underline{0} + \underline{0}, \{0 \mapsto (d_{\text{in},0}, z'), 1 \mapsto (d_{\text{in},1}, z')\}), 12)\})$$

where $25 = (\text{fst}(x) + \text{snd}(x)) \cdot 1$ and $12 = \text{fst}(x) \cdot 1$. This result makes sense because the last expression in the term t is '$x \cdot z$', so its backpropagator directly contributes cotangents to (1.) the input x with a partial derivative of 25, and (2.) the intermediate value $z = x + y$ with a partial derivative of 12. The sums $\underline{0} + \underline{0}$, of course, directly reduce to $\underline{0}$ instead of staying unevaluated, but we left them as-is to show what computation is happening in $(+_{\text{Staged}})$.

6.2 Remaining complexity challenges



We have gained a lot with the function call staging so far: where the naive algorithm from Section 5 easily ran into exponential blowup of computation time if the results of primitive operations were used in multiple places, the updated algorithm from Figure 8 completely solves this issue. For example, the program of Figure 2 now results in the call graph displayed on the right: each backpropagator is called exactly once. However, some other complexity problems still remain.[27]

As discussed in Section 3, 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. To account for programs that ignore their input, we add an additional term that P' may also read the full input regardless of whether P did so: the program $P = (\lambda(x : \tau).\ x) : \tau \to \tau$ always takes constant time whereas its gradient program must at the very least construct the value of P's full gradient, which has size size(x). Hence, we require that:

$$\exists c > 0.\ \forall P \in \text{Programs}(\sigma \to \tau).\ \forall x : \sigma, dy : \underline{\tau}. \\ \text{cost}(\text{snd}(\text{Wrap}[P]\ x)\ dy) \leq c \cdot (\text{cost}(P\ x) + \text{size}(x))$$

where cost(E) is the time taken to evaluate E, and size(x) is the time taken to read all of x sequentially.

So, what is cost(snd(Wrap[P] x) dy)? First, the primal pass (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.

[27] 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^4 after the improvements of Sections 7, 8 and 9.

https://doi.org/10.1017/S0956796825100051 Published online by Cambridge University Press

Summarising, the primal pass as a whole runs in $O(\text{cost}(P\,x) + \text{size}(x))$, which is precisely 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. $P = \lambda(x : \sigma).\ t$).

- 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 Figure 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 (not yet—see below), 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 Figure 8) is not constant-time: it adds values of type c and takes the union of staging maps, both of which may be expensive.
- Afterwards, we use SResolve on the resulting $\text{Staged}\,\sigma$ 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[28] 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. $(\underline{\lambda}(z : \underline{\mathbb{R}}).\ \text{SCotan}\ (0, \ldots, 0, z, 0, \ldots, 0))$ created in Interleave^2 (with SCotan from Wrap^2).
     Problem: The interleave backpropagators take time $O(\text{size}(x))$, not $O(1)$.
  2. $(\underline{\lambda}(z : \underline{\mathbb{R}}).\ 0_{\text{Staged}})$ created in $\mathbf{D}^2_\sigma[r]$.
     Problem: 0_{Staged} takes time $O(\text{size}(x))$, not $O(1)$.
  3. $(\underline{\lambda}(z : \underline{\mathbb{R}}).\ \text{SCall}\ d_1\ (\partial_1 op(...)(z)) +_{\text{Staged}} \cdots +_{\text{Staged}} \text{SCall}\ d_n\ (\partial_n op(...)(z)))$, as created in $\mathbf{D}^2_\sigma[op(...)]$. Assuming that primitive operation arity is bounded, we are allowed a constant-time operation for each argument to op.
     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.)

[28] 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.

https://doi.org/10.1017/S0956796825100051 Published online by Cambridge University Press

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 7) will be to Cayley-transform the Staged c object, including the contained c value, turning zero into the identity function 'id' and plus into function composition $(\circ)$ 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 7, 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 8 by replacing the c in Staged c with a more efficient structure (namely, Map $\mathbb{Z}\,\mathbb{R}$). This log-factor can optionally be further eliminated using mutable arrays as described in Section 9.

(C) The Map operations in SResolve are logarithmic in the size of the staging map. Like in the previous point, mutable arrays (Section 9) can eliminate this final log-factor in the complexity.

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.

7 Cayley-transforming the cotangent collector

For any monoid $(M, 0, +)$ we have a function $C_M : M \to (M \to M)$, given by $m \mapsto (m' \mapsto m + m')$. In fact, due to the associativity and unitality laws for monoids, this function defines a monoid homomorphism from $(M, 0, +)$ to the endomorphism monoid $(M \to M, \mathrm{id}, \circ)$ on M. By observing that C_M has a left-inverse $(C_M(m)(0) = m)$, we see that it even defines an isomorphism of monoids to its image, a fact commonly referred to as Cayley's theorem. This trick of realising a monoid M as a submonoid of its endomorphism monoid $M \to M$ is surprisingly useful in functional programming, as the operations of $(M \to M, \mathrm{id}, \circ)$ may have more desirable operational/complexity characteristics than the operations of $(M, 0, +)$. The optimisation discussed in this section is to switch to this *Cayley-transformed* representation for cotangents.

This trick is often known as the "difference list" trick in functional programming, due to its original application to improving the performance of repeated application of the list-append operation (Hughes, 1986). The intent of moving from $[\tau]$ (i.e. lists of values of type τ) to $[\tau] \to [\tau]$ 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 $(+_{\text{Staged}})$). 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 $(+_{\text{Staged}})$ turn into $(\circ)$. Since id is a value (of type Staged $c \to$ Staged c) 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.

https://doi.org/10.1017/S0956796825100051 Published online by Cambridge University Press

Of course, a priori this just moves the work to the primitive monoid values, which now have to update an existing value instead of directly returning a small value themselves. Because $M \to M$ is essentially a kind of sparse representation, however, this can sometimes be done more efficiently than with a separate addition operation.

After the Cayley transform, 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 primitive Staged c values, that is to say: the implementation of 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:[29] (the changed parts are shown in red)

$$\begin{aligned}
&\text{SCall} &&: (\mathbb{Z}, \underline{\mathbb{R}} \multimap (\text{Staged}\, c \to \text{Staged}\, c)) \to \underline{\mathbb{R}} \multimap (\text{Staged}\, c \to \text{Staged}\, c)\\
&\text{SCotan} &&: c \multimap (\text{Staged}\, c \to \text{Staged}\, c)\\
&\text{SResolve} &&: (\text{Staged}\, c \to \text{Staged}\, c) \multimap c
\end{aligned}$$

The definition of Staged c itself also gets changed accordingly:

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

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

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

Note that $(+_{\text{Staged}})$ 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:

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

Before calling SResolve, we simply apply the $(\text{Staged}\, c \to \text{Staged}\, c)$ function to 0_{Staged} (undoing the Cayley transform by using its left-inverse as discussed—this is now the only remaining 0_{Staged}); SResolve is then as it was in Section 6, only changing $f\ a +_{\text{Staged}} (c, m')$ to $f\ a\ (c, m')$ on the last line: f from the Map now has type $\underline{\mathbb{R}} \multimap (\text{Staged}\, c \to \text{Staged}\, c)$.

7.1 Code transformation

We show the new code transformation in Figure 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 on 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.

[29] Despite the fact that we forbade it in Section 5.1, 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 $(\text{Staged}\, c \to \text{Staged}\, c)$ corresponds with the sum (with $(+_{\text{Staged}})$) of the Staged c values that they represent.

https://doi.org/10.1017/S0956796825100051 Published online by Cambridge University Press

On types:

$\mathbf{D}^3_c[\mathbb{R}] = (\mathbb{R}, (\mathbb{Z}, \underline{\mathbb{R}} \multimap (\text{Staged } c \to \text{Staged } c)))$ $\quad \mathbf{D}^3_c[\mathbb{Z}] = \mathbb{Z}$ $\quad \mathbf{D}^3_c[()] = ()$

$\mathbf{D}^3_c[\sigma \to \tau] = \mathbf{D}^3_c[\sigma] \to \mathbb{Z} \to (\mathbf{D}^3_c[\tau], \mathbb{Z})$ $\quad \mathbf{D}^3_c[(\sigma, \tau)] = (\mathbf{D}^3_c[\sigma], \mathbf{D}^3_c[\tau])$

On terms:

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

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

Changed wrapper:

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

$\text{Deinterleave}^3_\tau : \forall c.\ \mathbf{D}^3_c[\tau] \to (\tau, \tau \multimap (\text{Staged } c \to \text{Staged } c))$

(Same as Deinterleave^2 in Fig. 8, except with id and $(\circ)$ in place of 0_{Staged} and $(+_{\text{Staged}})$)

$\text{Wrap}^3 : (\sigma \to \tau) \rightsquigarrow (\sigma \to (\tau, \tau \multimap \sigma))$

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

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

7.2 Running example

Apart from types, the only change in $\mathbf{D}^3_c$ since Section 6.1 is replacement of $(+_{\text{Staged}})$ by $(\circ)$. The result of $\text{Interleave}^3_{(\mathbb{R},\mathbb{R})}\ (x, \text{SCotan})\ 0$ is the same as before, except that the input backpropagators now update a Staged $(\mathbb{R}, \mathbb{R})$ pair instead of constructing one: $d_{\text{in},0} = \underline{\lambda}(z : \underline{\mathbb{R}}).\ \lambda((dx, dy), m).\ ((z + dx, dy), m)$ and $d_{\text{in},1} = \underline{\lambda}(z : \underline{\mathbb{R}}).\ \lambda((dx, dy), m).\ ((dx, z + dy), m)$. The Staged $(\mathbb{R}, \mathbb{R})$ object passed to SResolve is now:

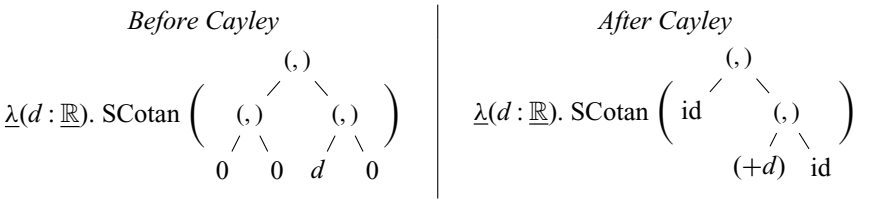
$$(\underline{0}, \{0 \mapsto (d_{\text{in},0}, 25), 2 \mapsto (\underline{\lambda}(z' : \underline{\mathbb{R}}).\ \lambda(c, m).\ (c, f\ m), 12)\})$$

where f is a function that adds the key-value pairs $0 \mapsto (d_{\text{in},0}, z')$ and $1 \mapsto (d_{\text{in},1}, z')$ into its argument m, inserting if not yet present and adding the second components of the values if they are. In this case, because the item at ID 2 (the backpropagator for $z = x + y$) is the first to be resolved by SResolve, this f will be passed an empty map, so the two pairs will be inserted.

https://doi.org/10.1017/S0956796825100051 Published online by Cambridge University Press

7.3 Remaining complexity challenges

In Section 6.2, 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 Wrap^3). 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 $\sigma = ((\mathbb{R}, \mathbb{Z}), (\mathbb{R}, \mathbb{R}))$, 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 8, essentially reducing issue two to issue three. Afterwards, in Section 9, we finally properly resolve issue three (logarithmic overhead from Map operations) by introducing mutable arrays. Doing so removes the final log-factors from the algorithm's complexity.

8 Keeping just the scalars: Efficient gradient updates

The codomain of the backpropagators is currently Staged $c \to$ Staged c, where Staged c is defined as:

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

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 $\underline{\mathbb{R}}$) that they receive in their argument, to a particular scalar in the collector.

Hence, all we need of c in Staged c is really the collection of its scalars: the rest simply came from $\underline{0} : c$ and is never changed.[30] 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 Map $\mathbb{Z}\ \underline{\mathbb{R}}$:

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

[30] If c contains coproducts (sum types) as we discuss in Section 11, this $\underline{0} : c$ becomes dependent on the actual input to the program, copying the structure from there.

https://doi.org/10.1017/S0956796825100051 Published online by Cambridge University Press

To accomodate this, Interleave changes to number all the scalars in the input with distinct IDs (conveniently with the same IDs as their corresponding input backpropagators, but this is no fundamental requirement); 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 Map $\mathbb{Z}\ \underline{\mathbb{R}}$) in time logarithmic in the size of the input. To be able to construct the final gradient from this collection of just its scalars, Interleave_τ additionally builds a *reconstruction* function of type $(\mathbb{Z} \to \underline{\mathbb{R}}) \to \tau$, which we pass a function that looks up the ID in the final collector Map to compute the actual gradient value.

Complexity. Now that we have fixed (in Section 7) the first complexity problem identified in Section 6.2 (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 Sections 3 and 6.2 apart from log-factors. More precisely, if we set $T = \text{cost}(P\ x)$ and $I = \text{size}(x)$, then $\text{Wrap}[P]$ computes the gradient of P at x not in the desired time $O(T + I)$ but instead in time $O\big(T \max(\log(T), \log(I)) + I \log(I)\big)$.[31] If we accept logarithmic overhead, we could choose to stop here. However, if we wish to strictly conform to the required complexity, or if we desire to lose the non-negligible constant-factor overhead of dealing with a persistent tree-map, we need to make the input backpropagators and Map operations in SResolve constant-time; we do this using mutable arrays in Section 9.

9 Using mutable arrays to shave off log factors

The analysis in Section 6.2 showed that after the Cayley transform in Section 7, the strict complexity requirements are met if we make the input backpropagators constant-time and make SResolve have only constant overhead for each backpropagator that it calls. Luckily, in both cases, the only component that is not constant-time is the interaction with one of the Maps in Staged c:

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

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:

$$\text{Staged}\ c = (\text{Array}\ \ \underline{\mathbb{R}}, \text{Array}\ \ (\underline{\mathbb{R}} \multimap (\text{Staged}\ c \to \text{Staged}\ c), \underline{\mathbb{R}}))$$

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 Wrap, 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

[31] There are $O(T)$ backpropagators to resolve, each of which could either modify the staging map (of size $O(T)$) or the gradient collector map (of size $O(I)$). (De)interleaving then does $O(n \log(n))$ work on the input of size $O(I)$.

https://doi.org/10.1017/S0956796825100051 Published online by Cambridge University Press

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.[32]

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 (of type Staged $c \to$ Staged c) produced by backpropagators, and in SResolve. Hence, in these two places, we need an effectful function type that allows us to encapsulate the mutability and ensure that it is not visible from the outside; options include a resource-linear function type and a monad for local side-effects such as the `ST` monad in Haskell. In this paper, we use a side-effectful monad; for a presentation of the sequential algorithm in terms of resource-linear types, see (Smeding and Vákár, 2022, Appendix A).

Time complexity. We now satisfy all the requirements of the analysis in Section 6.2, 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. Let $\mathbf{D}^4$, Interleave^4, Deinterleave^4, etc. be the definitions that use arrays as described above (see Section 9.1 for details). Then we can observe the following:

- The number of operations performed by $\mathbf{D}^4_c[t]$ (with the improvements from Sections 8 and 9) is only a constant factor times the number of operations performed by t, and hence in $O(T)$. This was already observed for $\mathbf{D}^2_c[t]$ in Section 6.2, and still holds.
- The number of backpropagators created while executing $\mathbf{D}^4_c[t]$ 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}), ($\circ$) (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 thanks to our staging, 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^4 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 $\underline{\mathbb{R}}$ in Staged c ($O(I)$); all this work is in $O(T + I)$.

Hence, calling Wrap$^4[t]$ 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 the size of the argument itself. This is indeed the correct time complexity for an efficient reverse AD algorithm, as discussed in Section 3.

[32] For worst-case programs, the space complexity of reverse AD is equal to the time complexity of the original program (Griewank and Walther, 2008). Reducing this space complexity comes at a trade-off to time complexity, using checkpointing (e.g. Siskind and Pearlmutter, 2018).

https://doi.org/10.1017/S0956796825100051 Published online by Cambridge University Press

9.1 Implementation using mutable references in a monad

As a purely functional language, Haskell chooses to disallow, in most parts of a program, any behavior that breaks referential transparency, including computational effects like mutable state: the languages forces the programmer to encapsulate such "dangerous" effectful behavior, when it is truly desired, in a monad, thus using the type system to isolate it from the pure code. The result is that the compiler can aggressively optimize the pure parts of the code while mostly leaving the effectful code, where correctness of optimizations is much more subtle, as is.

In particular, a typical design for mutable arrays in a purely functional language like Haskell is to use mutable references inside some monad. In Haskell, one popular solution is to use the `ST` monad (Launchbury and Jones, 1994) together with a mutable array library that exposes an API using `ST`, such as `STVector` in the `vector`[33] library. Because the `ST` monad is designed to be deterministic, it has a pure handler:

$$\mathrm{runST} : (\forall s.\ \mathtt{ST}\ s\ \alpha) \to \alpha$$

allowing the use of local mutability without compromising referential transparency of the rest of the program.[34]

However, precisely because of this design, `ST` does not support parallelism. (Parallelism in the presence of mutable references trivially allows non-deterministic behaviour.) For this reason, we will write the definitions from this point on in terms of `IO`, Haskell's catch-all monad for impurity. Fortunately, the only functionality we use from `IO` is parallelism and mutable arrays and references, and furthermore the design of our algorithm is such that the result is, in fact, deterministic even when the source program includes parallelism.[35] Thus, we can justify using $\mathtt{unsafePerformIO} :: \mathtt{IO}\ \alpha \to \alpha$ around the differentiated program, making the interface to the differentiated program pure again.

Letting the updater functions run in `IO` changes Staged c as follows:

$$\mathrm{Staged}\, c = (\mathrm{Array}\ \underline{\mathbb{R}}, \mathrm{Array}\ (\underline{\mathbb{R}} \multimap (\mathrm{Staged}\, c \to \mathtt{IO}\ ()), \underline{\mathbb{R}}))$$

for a suitable definition of Array, such as `IOVector` from `vector`. To write this definition, we need to give a monoid structure on Staged $c \to \mathtt{IO}\ ()$; fortunately, the only reasonable one ($f + g = \lambda x.\ f\ x \gg g\ x$) corresponds to the monoid structure on Staged c and is therefore precisely the one we want.[36]

Note that this definition now no longer structurally depends on c! This is to be expected, because the information about the structure of c is now contained in the *length* of the first array in a Staged c. For uniformity of notation, however, we will continue to write the c parameter to Staged.

[33] https://hackage.haskell.org/package/vector-0.13.1.0/docs/Data-Vector-Mutable.html

[34] The s parameter is informationless and only there to ensure correct scoping of mutable references in `ST`. For more info, see Launchbury and Jones (1994, §2.4), or Jacobs et al. (2022) for a formalised proof.

[35] From a theoretical perspective, this determinism follows from the fact that we do not actually use unrestricted mutation, but only *accumulation*—and accumulation is a commutative effect. In practice, however, the claim is technically untrue, because floating-point arithmetic is not associative. Given the nature of the computations involved, however, we still think getting parallelism is worth this caveat.

[36] $(\gg) :: \mathrm{Monad}\ m \Rightarrow m\ \alpha \to m\ \beta \to m\ \beta.\ m_1 \gg m_2$ runs both computations, discarding the result of m_1.

https://doi.org/10.1017/S0956796825100051 Published online by Cambridge University Press

Mutable arrays interface. We assume an interface to mutable arrays that is similar to that for `IOVector` in the Haskell `vector` library, cited earlier. In summary, we assume the following functions:

$$\begin{aligned}
&\texttt{alloc} &&: \mathbb{Z} \to \alpha \to \texttt{IO}\,(\text{Array}\,\alpha)\\
&\texttt{get} &&: \mathbb{Z} \to \text{Array}\,\alpha \to \texttt{IO}\,\alpha\\
&\texttt{modify} &&: \mathbb{Z} \to (\alpha \to \alpha) \to \text{Array}\,\alpha \to \texttt{IO}\,()\\
&\texttt{freeze} &&: \text{Array}\,\alpha \to \texttt{IO}\,(\text{IArray}\,\alpha)\\
&(@) &&: \text{IArray}\,\alpha \to \mathbb{Z} \to \alpha
\end{aligned}$$

where IArray is an immutable array type. `alloc` and `freeze` are linear in the length of the array; `get`, `modify` and (@) are constant-time (in addition to calling the function once, of course, for `modify`).

Let us see how the code transformation changes with mutable arrays.

Code transformation. Using the new definition of Staged, we change the code transformation once more, this time from Figure 9 to the version given in Figure 10. The transformation on types and terms simply sees the type of scalar backpropagators change to use effectful updating instead of functional updating, so they do not materially change: we just give yet another interpretation of 0_{Staged} and $(+_{\text{Staged}})$, using the monoid structure on Staged $c \to \texttt{IO}\,()$ defined above in terms of $(\gg)$. However, some important changes occur in the Staged c interface and the wrapper. Let us first look at the algorithm from the top, by starting with Wrap^4; after understanding the high-level idea, we explain how the other components work.

In basis, Wrap^4 does the same as Wrap^3 from Figure 9: interleave injector backpropagators with the input of type σ, execute the transformed function body using the interleaved input, and then deinterleave the result. However, because we (since Section 8) represent the final cotangent not directly as a value of type σ in a Staged σ but instead as an array of only the embedded scalars (Array $\mathbb{R}$), some more work needs to be done.

First, $\text{Interleave}^4_\sigma$ (monadically, in the ID generation monad that we are writing out explicitly) produces, in addition to the interleaved input, also a *reconstruction* function[37] of type IArray $\underline{\mathbb{R}} \to \sigma$. This rebuilder takes an array with precisely as many scalars as were in the input, and produces a value of type σ with the structure (and discrete-typed values) of the input, but the scalars from the array. The mapping between locations in σ and indices in the array is the same as the numbering performed by Interleave^4.

Having x, *rebuild* and i (the next available ID), we execute the transformed term $\mathbf{D}^4_\sigma[\![t]\!]$ monadically (with x in scope), resulting in an output $y' : \mathbf{D}^4_\sigma[\![\tau]\!]$. This output we deinterleave to $y : \tau$ and $d : \tau \multimap (\text{Staged}\,\sigma \to \texttt{IO}\,())$.

The final result then consists of the regular function result (y) as well as the top-level derivative function of type $\tau \multimap \sigma$. In the derivative function, we allocate two arrays to initialise an empty Staged σ (note that the given sizes are indeed precisely large enough), and apply d to the incoming τ cotangent. This gives us an updater function that (because of how Deinterleave^4 works) calls the top-level backpropagators contained in y' in the Staged arrays, and we apply this function to the just-allocated Staged object. Then we use

[37] Implementing the $(\mathbb{Z} \to \underline{\mathbb{R}})$ in $(\mathbb{Z} \to \underline{\mathbb{R}}) \to \tau$ from Section 8 as IArray $\underline{\mathbb{R}}$.

https://doi.org/10.1017/S0956796825100051 Published online by Cambridge University Press

On types:

$$\mathbf{D}^4_c[\mathbb{R}] = (\mathbb{R}, (\mathbb{Z}, \underline{\mathbb{R}} \multimap (\text{Staged } c \to \texttt{IO}\ ()))) \qquad \mathbf{D}^4_c[\mathbb{Z}] = \mathbb{Z} \qquad \mathbf{D}^4_c[()] = ()$$
$$\mathbf{D}^4_c[\sigma \to \tau] = \mathbf{D}^4_c[\sigma] \to \mathbb{Z} \to (\mathbf{D}^4_c[\tau], \mathbb{Z}) \qquad \mathbf{D}^4_c[(\sigma, \tau)] = (\mathbf{D}^4_c[\sigma], \mathbf{D}^4_c[\tau])$$

On terms:

If $\Gamma \vdash t : \tau$ then $\mathbf{D}^4_c[\Gamma] \vdash \mathbf{D}^4_c[t] : \mathbb{Z} \to (\mathbf{D}^4_c[\tau], \mathbb{Z})$

Same as $\mathbf{D}^2_c$, except with '$\lambda\_.$ **return** ()' in place of 0_{Staged} and '$\lambda f\ g\ x.\ f\ x \gg g\ x$' in place of $(+_{\text{Staged}})$.

New Staged interface:

$$\text{Staged } c = (\text{Array } \underline{\mathbb{R}}, \text{Array } (\underline{\mathbb{R}} \multimap (\text{Staged } c \to \texttt{IO}\ ()), \underline{\mathbb{R}}))$$

$$\text{SAlloc} : \mathbb{Z} \to \mathbb{Z} \to \texttt{IO}\ (\text{Staged } c)$$
$$\text{SAlloc } i_{\text{inp}}\ i_{\text{out}} = \texttt{alloc}\ i_{\text{inp}}\ 0 \gg\!= \lambda c.\ \texttt{alloc}\ i_{\text{out}}\ (\underline{\lambda}(z : \underline{\mathbb{R}}).\ \text{id}, 0) \gg\!= \lambda m.\ \textbf{return}\ (c, m)$$

$$\text{SCall} : (\mathbb{Z}, \underline{\mathbb{R}} \multimap (\text{Staged } c \to \texttt{IO}\ ())) \to \underline{\mathbb{R}} \multimap (\text{Staged } c \to \texttt{IO}\ ())$$
$$\text{SCall}\ (i, f)\ a\ (c, m) = \texttt{modify}\ i\ (\lambda(\_, a').\ (f, a + a'))\ m$$

$$\text{SOneHot} : \mathbb{Z} \to \underline{\mathbb{R}} \multimap (\text{Staged } c \to \texttt{IO}\ ())$$
$$\text{SOneHot}\ i\ a\ (c, m) = \texttt{modify}\ i\ (\lambda(a' : \mathbb{R}).\ a + a')\ c$$

$$\text{SResolve} : \mathbb{Z} \to \text{Staged } c \to \texttt{IO}\ (\text{IArray } \underline{\mathbb{R}})$$
$$\text{SResolve}\ i_{\text{out}}\ (c, m) = \mathit{loop}\ (i_{\text{out}} - 1) \gg \texttt{freeze}\ c$$
$$\text{where } \mathit{loop}\ (-1) = \textbf{return}\ ()$$
$$\mathit{loop}\ i \quad = \texttt{get}\ i\ m \gg\!= \lambda(f, a).\ f\ a\ (c, m) \gg \mathit{loop}\ (i - 1)$$

Wrapper:

$$\text{Interleave}^4_\tau \quad : \tau \to \mathbb{Z} \to ((\mathbf{D}^4_c[\tau], \text{IArray } \underline{\mathbb{R}} \to \tau), \mathbb{Z})$$
$$\text{Interleave}^4_{\mathbb{R}} \quad = \lambda x.\ \lambda i.\ (((x, (i, \text{SOneHot } i)), \lambda a.\ a\ @\ i), i + 1)$$
$$\text{Interleave}^4_{()} \quad = \lambda().\ \lambda i.\ (((), \lambda a.\ ()), i)$$
$$\text{Interleave}^4_{(\sigma,\tau)} = \lambda(x, y).\ \lambda i.\ \textbf{let}\ ((x', f_1), i') = \text{Interleave}^4_\sigma\ x\ i$$
$$\textbf{in let}\ ((y', f_2), i'') = \text{Interleave}^4_\tau\ y\ i'$$
$$\textbf{in}\ (((x', y'), \lambda a.\ (f_1\ a, f_2\ a)), i'')$$
$$\text{Interleave}^4_{\mathbb{Z}} \quad = \lambda n.\ \lambda i.\ ((n, \lambda a.\ n), i)$$

$$\text{Deinterleave}^4_\tau : \mathbf{D}^4_c[\tau] \to (\tau, \tau \multimap (\text{Staged } c \to \texttt{IO}\ ()))$$
(Same as Deinterleave^2, except with the same monoid changes as $\mathbf{D}^4_c$ above)

$$\text{Wrap}^4 : (\sigma \to \tau) \rightsquigarrow (\sigma \to (\tau, \tau \multimap \sigma))$$
$$\text{Wrap}^4[\lambda(x : \sigma).\ t] = \lambda(x : \sigma).$$
$$\textbf{let}\ ((x : \mathbf{D}^4_\sigma[\sigma], \mathit{rebuild} : \text{IArray } \underline{\mathbb{R}} \to \sigma), i) = \text{Interleave}^4_\sigma\ x\ 0$$
$$\textbf{in let}\ (y', i') = \mathbf{D}^4_\sigma[t]\ i$$
$$\textbf{in let}\ (y, d : \tau \multimap (\text{Staged } \sigma \to \texttt{IO}\ ())) = \text{Deinterleave}^4_\tau\ y'$$
$$\textbf{in}\ (y, \underline{\lambda}(z : \tau).\ \mathit{rebuild}\ (\texttt{unsafePerformIO}$$
$$(\text{SAlloc } i\ i' \gg\!= \lambda s.\ d\ z\ s \gg \text{SResolve } i'\ s)))$$

Fig. 10. Code transformation plus wrapper using mutable arrays, modified from Figure 9. Grey parts are unchanged.

https://doi.org/10.1017/S0956796825100051 Published online by Cambridge University Press

the new SResolve to propagate the cotangent contributions backwards, by invoking each backpropagator in turn in descending order of IDs. Like before in the Cayley-transformed version of our AD technique, those backpropagators update the state (now mutably) to record their own contributions to (i.e. invocations of) other backpropagators. At the end of SResolve, the backpropagator staging array is dropped and the cotangent collection array is frozen and returned as an IArray $\underline{\mathbb{R}}$ (corresponding to the c value in a Staged c for the Cayley-transformed version in Figure 9).

This whole derivative computation is, in the end, pure (in that it is deterministic and has no side-effects), so we can safely evaluate the `IO` using `unsafePerformIO` to get a pure IArray $\underline{\mathbb{R}}$, which contains the scalar cotangents that *rebuild* from Interleave^4 needs to put in the correct locations in the input, thus constructing the final gradient.

Implementation of the components. Having discussed the high-level sequence of operations, let us briefly discuss the implementation of the Staged c interface and (de)interleaving. In Interleave^4, instead of passing structure information down in the form of a setter $((\tau \to \tau) \multimap (\text{Staged}\, c \to \text{Staged}\, c))$ like we did in Interleave^3 in Figure 9, we build structure information up in the form of a getter $(\text{IArray}\, \underline{\mathbb{R}} \to \tau)$. This results in a somewhat more compact presentation, but in some sense the same information is still communicated.

The program text of Deinterleave^4 is again unchanged, because it is agnostic about the codomain of the backpropagators, as long as it is a monoid, which it remains.

On the Staged interface, the transition to mutable arrays had a significant effect. The 0_{Staged} created by Wrap^3 in Section 7 is now essentially in SAlloc, which uses `alloc` to allocate a zero-filled cotangent collection array of size i_{inp}, and the backpropagator staging array of size i_{out} filled with zero-backpropagators with an accumulated argument of zero.

SCall has essentially the same type, but its implementation differs because it now performs a constant-time mutable update on the backpropagator staging array instead of a logarithmic-complexity immutable Map update. Note that, unlike in Section 7, there is no special case if i is not yet in the array, because unused positions are already filled with zeros.

SOneHot takes the place of SCotan, with the difference that we have specialised it using the knowledge that all relevant $c \to c$ functions add a particular scalar to a particular index in the input, and that these functions can hence be defunctionalised to a pair $(\mathbb{Z}, \underline{\mathbb{R}})$. The monoid-linearity here is in the real scalar, as it was before, hence the placement of the $\multimap$-arrow.

Finally, SResolve takes an additional $\mathbb{Z}$ argument that should contain the output ID of $\mathbf{D}_c^4[t]$, i.e. one more than the largest ID generated. *loop* then does what the original SResolve did directly, iterating over all IDs in descending order and applying the state updaters in the backpropagator staging array one-by-one to the state. After the loop is complete, we freeze and return just the cotangent collection array, because we have no need for the staging array any more. This frozen collection array will then be used to build the final gradient in Wrap^4.

https://doi.org/10.1017/S0956796825100051 Published online by Cambridge University Press

10 Was it taping all along?

In this section we first apply one more optimisation to our algorithm to make it slightly more efficient (Section 10.1). Afterwards, we show that defunctionalising the backpropagators (Section 10.2) essentially reduces the technique to classical taping approaches (Section 10.3).

10.1 Dropping the cotangent collection array

Recall that the final transformation of Section 9 used two mutable arrays threaded through the backpropagators in the Staged c pair: a cotangent collection array of type Array $\underline{\mathbb{R}}$ and a backpropagator call staging array of type Array $(\underline{\mathbb{R}} \multimap (\text{Staged } c \to \texttt{IO}\ ()), \underline{\mathbb{R}})$, using the monad-based implementation from Section 9.1. 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 $\text{Interleave}_{\mathbb{R}}$ 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 we have $c[i] = \text{snd}(m[i])$ for all i for which $c[i]$ is defined, if (c, m) is the input to SResolve for which the recursion terminates. Therefore, the cotangent collection array is actually unnecessary: its information can be read off directly from the backpropagator staging array.

With this knowledge, we can instead use Staged $c = \text{Array}\ (\underline{\mathbb{R}} \multimap (\text{Staged } c \to \texttt{IO}\ ()), \underline{\mathbb{R}})$ as our definition. The reconstruction functions of Section 8 simply take the second projection of the correponding array element.

10.2 Defunctionalisation of backpropagators

In the core code transformation ($\mathbf{D}_c$, excluding the wrapper), all backpropagators are (now) of type $\underline{\mathbb{R}} \multimap (\text{Staged } c \to \texttt{IO}\ ())$, and, as observed earlier in Section 6, these backpropagators come in only a limited number of forms:

1. The input backpropagators created in $\text{Interleave}_{\mathbb{R}}$, reduced to $(\underline{\lambda}(z : \underline{\mathbb{R}}).\ \mathbf{return}\ ())$ in Section 10.1;
2. $(\underline{\lambda}(z : \underline{\mathbb{R}}).\ \mathbf{return}\ ())$ created in $\mathbf{D}^4_c[r]$ for scalar constants r;
3. $(\underline{\lambda}(z : \underline{\mathbb{R}}).\ \text{SCall}\ d_1\ (\partial_1 op(x_1, \ldots, x_n)(z)) \circ \cdots \circ \text{SCall}\ d_n\ (\partial_n op(x_1, \ldots, x_n)(z)))$ 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

$$\partial_i op(x_1, \ldots, x_n)(z) = z \cdot \partial_i op(x_1, \ldots, x_n)(1)$$

Hence, we can defunctionalise (Reynolds, 1998) and change all occurrences of the type $\underline{\mathbb{R}} \multimap (\text{Staged } c \to \texttt{IO}\ ())$ to Contrib, where $\text{Contrib} = [(\underline{\mathbb{R}}, (\mathbb{Z}, \text{Contrib}))]$: a list of triples of a scalar, an integer ID, and a recursive Contrib structure. The ID is the ID of the Contrib structure (i.e. the backpropagator) that it is adjacent to. (As before, these IDs make sharing observable.) In this representation, we think of $[(a_1, (i_1, cb_1)), \ldots, (a_n, (i_n, cb_n))]$ of type

https://doi.org/10.1017/S0956796825100051 Published online by Cambridge University Press

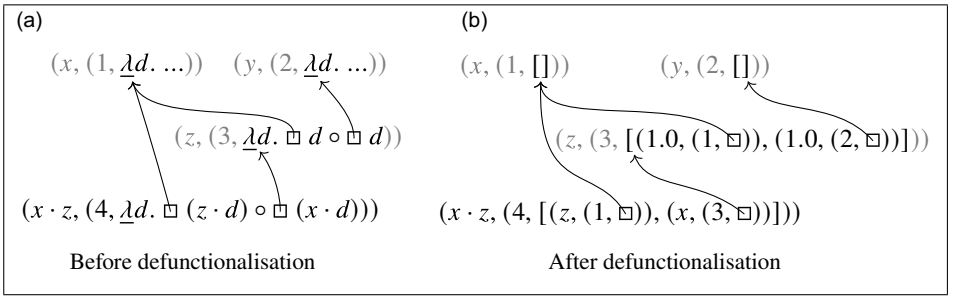


Fig. 11. The sharing structure before and after defunctionalisation. SCall is elided here; in Figure 11(a), the backpropagator calls are depicted as if they are still normal calls. Boxes (□) are the same in-memory value as the value their arrow points to; two boxes pointing to the same value indicates that this value is *shared*: referenced in two places.

Contrib as the backpropagator

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

For example, suppose we differentiate the running example program:

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

using the final algorithm of Section 9.1. The return value from the $\mathbf{D}_{(\mathbb{R},\mathbb{R})}$-transformed code (when applied to the output from $\mathrm{Interleave}_{(\mathbb{R},\mathbb{R})}$) has the sharing structure shown in Figure 11(a). This shows how the backpropagators refer to each other in their closures.

If we perform the type replacement (Section 10.1) and the defunctionalisation (this subsection), Interleave simplifies and SCall disappears; backpropagators of forms (1) and (2) become [] (the empty list) and those of form (3) become:

$$[(\partial_1 op(x_1, \ldots, x_n)(1), d_1), \ldots, (\partial_n op(x_1, \ldots, x_n)(1), d_n)]$$

SResolve then interprets a list of such $(a, (i, cb))$ by iterating over the list and for each such triple, replacing (cb', a') at index i in the staging array with $(cb, a' + a \cdot d)$, where d is the cotangent recorded in the array cell where the list was found.

10.3 Was it taping all along?

After the improvements from Sections 10.1 and 10.2, what previously was a tree of (staged) calls to backpropagator functions is now a tree of Contrib values with attached IDs[38] 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.

[38] Note that we now have $\mathbf{D}[\mathbb{R}] = (\mathbb{R}, (\mathbb{Z}, \mathrm{Contrib}))$, the integer being the ID of the Contrib value.

https://doi.org/10.1017/S0956796825100051 Published online by Cambridge University Press

This means that we can already compute the Contrib lists and write them to the array in the forward pass, if we change the ID generation monad that the differentiated code already lives in (which is a state monad with a single $\mathbb{Z}$ as state) to additionally carry the staging array, and furthermore change the monad to thread its state in a way that allows mutation, again using the techniques from Section 9, but now in the forward pass too. 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) to which we write the performed operations 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 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)).

11 Extending the source language

The source language (Figure 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 correctness proof for the algorithm can be extended to many expressive language constructs in the source language, such as coproducts and recursive types by using suitable logical relations arguments (Lucatelli Nunes and Vákár, 2024). Further, 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 the (reverse of the) linear sequence of primitive operations on scalars that was performed in the particular execution of the forward pass.

(Mutual) recursion. For example, we can allow recursive functions in our source language by adding recursive let-bindings with the following typing rule:

$$\frac{\Gamma, f : \sigma \to \tau, x : \sigma \vdash s : \tau \qquad \Gamma, f : \sigma \to \tau \vdash t : \rho}{\Gamma \vdash \mathbf{letrec}\, f\ x = s\ \mathbf{in}\, f\ t : \rho}$$

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. Note that due to the assumed call-by-value semantics, recursive non-function definitions would make little sense.

Recursion introduces the possibility of non-termination because the reverse pass is nothing more than a loop over the primitive scalar operations performed in the forward

https://doi.org/10.1017/S0956796825100051 Published online by Cambridge University Press

execution, the derivative program terminates exactly if the original program terminates (on a machine with sufficient memory).

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 ($\sigma, \tau ::= \cdots \mid \sigma \sqcup \tau$) 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 is in scope in t_1 and y is in scope in t_2. Then the type and code transformations extend in the unique structure-preserving manner:

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

To create an interesting interaction between control flow and differentiation, we can add a construct 'sign' with the unsurprising typing rule

$$\frac{\Gamma \vdash t : \mathbb{R}}{\Gamma \vdash \mathrm{sign}(t) : \mathrm{Bool}}$$

where $\mathrm{Bool} = () \sqcup ()$, which allows us to perform a case distinction on the sign of a real number. For differentiation of this construct, it suffices to define $\mathbf{D}_c^1[\mathrm{sign}(t)] = \mathrm{sign}(\mathrm{fst}(\mathbf{D}_c^1[t]))$.

If one accepts losing some of the structure-preserving nature of the transformation, it is possible to prevent redundant differentiation of t in $\mathbf{D}_c^1[\mathrm{sign}(t)]$ by making clever substitutions in t's free variables, converting back from dual-number form to the plain data representation. The idea is to define functions $\varphi_\tau : \mathbf{D}_c^1[\tau] \to \tau$ and $\psi_\tau : \tau \to \mathbf{D}_c^1[\tau]$ by induction on τ, where φ_τ projects out the value from a dual number and ψ_τ injects scalars into a dual number as constants (i.e. with a zero backpropagator). φ_τ and ψ_τ are mutually recursive at function types: e.g. $\varphi_{\sigma\to\tau}\ f = \varphi_\tau \circ f \circ \psi_\sigma$. Then one can define a non-differentiating transformation $\mathbf{D}_c^{\mathrm{plain}}$ on terms that uses φ to convert free variables to plain values, and otherwise keeps all code plain. We then get $\mathbf{D}_c^1[\mathrm{sign}(t)] = \mathrm{sign}(\mathbf{D}_c^{\mathrm{plain}}[t])$.

When moving to $\mathbf{D}_c^4$, the type transformation for coproducts stays unchanged, and the term definitions change only by transitioning 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.

The introduction of dynamic control flow complicates the correctness story for any AD algorithm. The approach presented here has the usual behaviour: derivatives are correct in the interior of domains leading to a particular execution path (if 'sign' is the only "continuous conditional", this is for inputs where none of the 'sign' operations receive zero as their arguments), but may be unexpected at the points of branching. For discussion see e.g. (Hückelheim et al., 2023, §3.3); for proofs see e.g. (Lucatelli Nunes and Vákár, 2024, §11) or Mazza and Pagani (2021).

https://doi.org/10.1017/S0956796825100051 Published online by Cambridge University Press

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

$$\textbf{data}\ T_1\ \alpha = C_1\ \alpha\ (T_2\ \alpha) \mid C_2\ \mathbb{R}$$
$$\textbf{data}\ T_2\ \alpha = C_3\ \mathbb{Z}\ (T_1\ \alpha)\ (T_2\ \alpha)$$

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

$$\textbf{data}\ DT_1\ \alpha = DC_1\ \alpha\ (DT_2\ \alpha) \mid DC_2\ (\mathbb{R}, \underline{\mathbb{R}} \multimap c)$$
$$\textbf{data}\ DT_2\ \alpha = DC_3\ \mathbb{Z}\ (DT_1\ \alpha)\ (DT_2\ \alpha)$$

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

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

Furthermore, for plain type variables, we set $\mathbf{D}^1_c[\alpha] = \alpha$.[39]

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 Figure 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 $\mathbf{D}^1_c[\alpha] = \alpha$, polymorphic functions can also be differentiated transparently, similarly to how the above handles polymorphic data types.

12 Parallelism

So far, we have assigned sequentially increasing IDs to backpropagators in the forward pass and resolved them in their linear order from top to bottom during the reverse pass. As long as the source program is executed sequentially, such ID generation is appropriate. However, if the source program uses parallelism in its execution, such linear ID assignment discards this parallelism structure and prevents us from exploiting it for computing the derivative in parallel.

In this section, we explore how to perform dual numbers reverse AD on source programs that contain fork-join task parallelism using a simple, but representative, parallel combination construct ($\star$) that has the semantics that $s \star t$ computes s and t in parallel and returns the pair (s, t). We discuss a different ID assignment scheme for backpropagators that takes parallelism into account, and we show that we can take advantage of these new IDs to resolve backpropagators in parallel during the reverse pass.

[39] 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.

https://doi.org/10.1017/S0956796825100051 Published online by Cambridge University Press

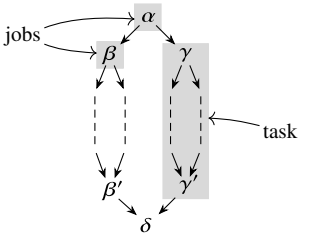


Fig. 12. Schematic view of the operational model underlying ($\star$).

12.1 Fork-Join parallelism

We work with a parallel operational model in the fork-join style. Roughly speaking: to run two subcomputations in parallel, a task *forks* into two sub-tasks; this pauses the parent task. After the sub-tasks are done, they *join* back into the parent task, which then resumes execution. A task may fork as many times as it likes. Each individual sequential section of execution (i.e. the part of a task before its first fork, between its join and the next fork, etc. and finally the part after the last join) we call a *job*. Each job in the program gets a fresh job ID that is its unique identifier. The intent is that independent jobs can execute in parallel on different CPU cores. A typical runtime for this model is a thread pool together with a job queue to distribute work over the operating system threads: new jobs are submitted to the queue when they are created, and threads from the pool pick up waiting jobs from the queue when idle.

Concretely, we extend our source language syntax with a parallel pairing construct ($\star$) with the following typing rule:

$$\frac{\Gamma \vdash s : \sigma \qquad \Gamma \vdash t : \tau}{\Gamma \vdash s \star t : (\sigma, \tau)}$$

Operationally, when we encounter the term $t_1 \star t_2$ while evaluating some term t in job α, two new jobs are created with fresh job IDs β and γ. The term t_1 starts evaluating in job β, potentially does some forks and joins, and finishes in a (potentially) different job β', returning a result v; t_2 starts evaluating in job γ and finishes in γ', returning a result w. When β' and γ' terminate, evaluation of t continues in a new job δ with the result (v, w) for $t_1 \star t_2$. We say that α *forks* into β and γ and that β' and γ' *join* into δ. The two parallel subgraphs, one from β to β' and one from γ to γ', we call *tasks*. The operational model does not know about tasks (it only knows about jobs), but we will use the concept of tasks in Section 12.3 as a compositional building block for the job graph. Figure 12 contains a diagrammatic representation of the preceding paragraph.

The algorithm in this section extends readily to n-ary parallel tupling constructs:

$$\frac{\Gamma \vdash t_1 : \tau_1 \quad \cdots \quad \Gamma \vdash t_n : \tau_n}{\Gamma \vdash (t_1, \ldots, t_n)^\star : (\tau_1, \ldots, \tau_n)}$$

or even parallel combination constructs of dynamic arity, where $[\tau]$ denotes lists of τ:

$$\frac{\Gamma \vdash t : [() \to \tau]}{\Gamma \vdash \star t : [\tau]}$$

but for simplicity of presentation, we restrict ourselves here to binary forking.

https://doi.org/10.1017/S0956796825100051 Published online by Cambridge University Press

12.2 Parallel IDs and their partial order

To avoid discarding the (explicit,[40] using ($\star$)) parallelism structure in the source program, we have to somehow record the dependency graph of the backpropagators (the same graph as the computation graph of scalars in the source program) in a way that is more precise than the chronological linear order used for the sequential algorithm. Specifically, we want backpropagators that were created in parallel jobs in the forward pass to not depend on each other in the dependency graph, not even transitively. In other words, they should be incomparable in the partial order that informs the reverse pass (SResolve) what backpropagator to resolve next.

To support the recording of this additional dependency information, we switch to *compound IDs*, consisting of two integers instead of one:

- The *job ID* that uniquely identifies the job a backpropagator is created from. This requires that we have a way to generate a unique ID in parallel every time a job forks or two jobs join. Job IDs do *not* carry a special partial order; see below.
- The *ID within the job*, which we assign sequentially (starting from 0) to all backpropagators created in a job. The operations within one forward-pass job are sequential (because of our fork-join model where a fork ends a job; see the previous subsection); this ID-within-job simply witnesses this.

Compound IDs have lexicographic order: $(\alpha, i) \leq_{\mathrm{c}} (\beta, i')$ iff $\alpha \leq_{\mathrm{j}} \beta \wedge (\alpha \neq \beta \vee i \leq_{\mathbb{Z}} i')$. The order on sequential IDs (within a job) is simply the standard linear order $\leq_{\mathbb{Z}}$, but the partial order on job IDs is different and is instead defined as the transitive closure of the following three cases:

1. $\alpha \leq_{\mathrm{j}} \alpha$;
2. If α forks into β and γ, then $\alpha \leq_{\mathrm{j}} \beta$ and $\alpha \leq_{\mathrm{j}} \gamma$;
3. If α and β join into γ, then $\alpha \leq_{\mathrm{j}} \gamma$ and $\beta \leq_{\mathrm{j}} \gamma$.

In Figure 13, we give an example term together with graphs showing the (generators of the) partial orders $\leq_{\mathrm{j}}$ on job IDs and $\leq_{\mathrm{c}}$ on compound IDs. Both graphs have arrows pointing to the successors of each node: n_2 is a successor of n_1 (and n_1 a predecessor of n_2) if $n_1 < n_2$ (i.e. $n_1 \leq n_2$ and $n_1 \neq n_2$) and there is no m such that $n_1 < m < n_2$. The graphs generate their respective partial orders if one takes the transitive closure and includes trivial self-loops. For Figure 13(b) ($\leq_{\mathrm{j}}$), the arrows thus show the fork/join relationships; for Figure 13(c) ($\leq_{\mathrm{c}}$), this is refined with the linear order on sequential IDs within each job. Note that these compound IDs replace the integer IDs of the sequential algorithm of Sections 5 to 10; that is to say: the result of every primitive operation gets a unique ID.

The reverse pass (SResolve) needs to traverse the dependency graph on compound IDs (Figure 13(c)) in reverse dependency order (taking advantage of task parallelism), but it is actually unnecessary to construct the full graph at runtime. It is sufficient to construct

[40] Because our source language is pure, one could in principle detect and exploit implicit parallelism. We focus on explicit parallelism here because automatic parallelism extraction is (difficult and) orthogonal to this work. In effect, the dependency graph that we construct is a weakening of the perfectly accurate one: computations within a job are assumed sequentially dependent. The reverse pass in Section 12.4 simply walks our constructed dependency graph, exploiting all apparent parallelism; SResolve there only inherits the concept of tasks and jobs because we encode the graph in a particular way that makes use of that structure (Section 12.3).

https://doi.org/10.1017/S0956796825100051 Published online by Cambridge University Press

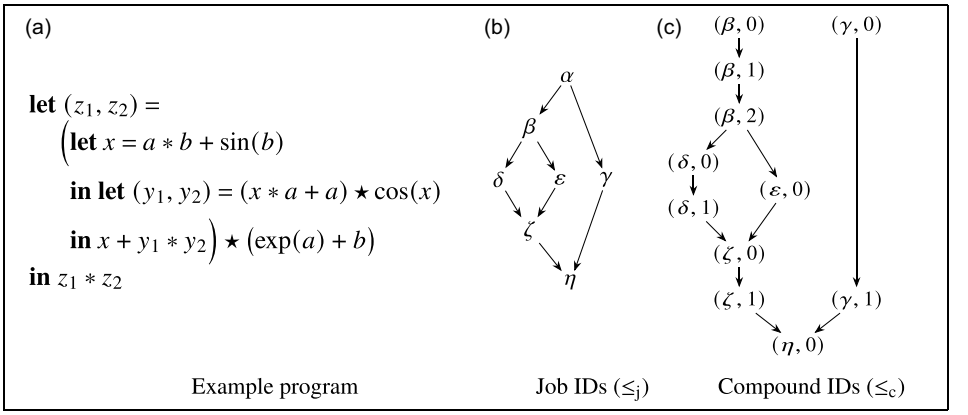


Fig. 13. An example program. Note that the program starts by forking, before performing any primitive operations, hence job α is empty and the partial order on compound IDs happens to have multiple minimal elements.

the dependency graph on *job IDs* (Figure 13(b)) together with, for each job α, the number n_α of sequential IDs generated in that job (note that this number may be zero if no scalar computation was done while running in that job). With this information, SResolve can walk the job graph in reverse topological order, resolving parallel tasks in parallel, and for each job α sequentially resolve the individual backpropagators from $n_\alpha - 1$ to 0. We will collect this additional information during the forward pass by extending the monad that the forward pass runs in.

12.3 Extending the monad

In the first optimisation that we applied to the (sequential) algorithm, namely linear factoring via staging of backpropagators (Section 6, Figure 8), we modified the code transformation to produce code that runs in an state monad $\mathcal{M}\,\tau = \mathrm{ID} \to (\tau, \mathrm{ID})$, for $\mathrm{ID} = \mathbb{Z}$:

$$\text{If } \Gamma \vdash t : \tau \text{ then } \mathbf{D}_c^2[\Gamma] \vdash \mathbf{D}_c^2[t] : \mathcal{M}\,\mathbf{D}_c^2[\tau]$$

In fact, this state monad was simply the natural implementation of an *ID generation monad* with one method:

$$\begin{aligned} &\mathrm{genID} \;:\; \mathcal{M}\,\mathrm{ID} \\ &\mathrm{genID} = \lambda(i : \mathbb{Z}).\ (i+1, i) \end{aligned}$$

We saw above in Section 12.2 that we need to extend this monad to be able to do two things: (1) generate compound IDs, not just sequential IDs and (2) record the job graph resulting from parallel execution using $(\star)$. Write JID for the type of job IDs (in an implementation we can simply set $\mathrm{JID} := \mathbb{Z}$) and write $\mathrm{CID} := (\mathrm{JID}, \mathbb{Z})$ for the type of compound IDs. The extended monad needs two methods:

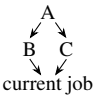
$$\begin{aligned} &\mathrm{genID} : \mathcal{M}\,\mathrm{CID} \\ &(\star') : \mathcal{M}\,\sigma \to \mathcal{M}\,\tau \to \mathcal{M}\,(\sigma, \tau) \end{aligned}$$

https://doi.org/10.1017/S0956796825100051 Published online by Cambridge University Press

$$\mathcal{M}\ \alpha = \texttt{IORef}\ \text{JID} \to \text{JobDescr} \to \texttt{IO}\ (\text{JobDescr}, \alpha)$$

$$\text{JobDescr} = (\text{History}, \text{JID}, \mathbb{Z})$$

$$\textbf{data}\ \text{History} = \text{Start} \mid \text{Fork}\ \underset{\text{(A)}}{\text{JobDescr}}\ \underset{\text{(B)}}{\text{JobDescr}}\ \underset{\text{(C)}}{\text{JobDescr}}$$



$$f \star' g = \lambda ref\ jd_0.\ \textbf{do}$$
$$(j_1, j_2, j_3) \leftarrow \texttt{atomicModifyIORef'}\ ref\ (\lambda j.\ (j+3, (j, j+1, j+2)))$$
$$((jd_1, x), (jd_2, y)) \leftarrow \texttt{inParallel}\ (f\ ref\ (\text{Start}, j_1, 0))\ (g\ ref\ (\text{Start}, j_2, 0))$$
$$\textbf{return}\ ((\text{Fork}\ jd_0\ jd_1\ jd_2, j_3, 0), (x, y))$$
$$\text{where: } \texttt{atomicModifyIORef'} :: \texttt{IORef}\ \alpha \to (\alpha \to (\alpha, \beta)) \to \texttt{IO}\ \beta$$
$$\texttt{inParallel} :: \texttt{IO}\ \alpha \to \texttt{IO}\ \beta \to \texttt{IO}\ (\alpha, \beta)$$

Fig. 14. Sketch of the implementation of the monad $\mathcal{M}$. The diagram shows the meaning of the job descriptions in "Fork": the first field (labeled "*A*") contains the history up to the last fork in this task (excluding subtasks), and the fields labeled *B* and *C* describe the subtasks spawned by that fork. The first job in a task has no history, indicated with "Start".

The updated genID reads the current job ID from reader context inside the monad and pairs that with a sequential ID generated in the standard fashion with monadic state. The monadic parallel combination method, ($\star'$), generates some fresh job IDs by incrementing an atomic mutable cell in the monad, runs the two monadic computations *in parallel* by spawning jobs as described in Section 12.1 and records the structure of the job graph thus created in some state inside the monad.

In an implementation, one can take the definitions in Figure 14. Working in Haskell, we write `IORef` for a mutable cell and use `IO` as the base monad in which we can access that mutable cell as well as spawn and join parallel threads. (We do not use `IO` in any other way in $\mathcal{M}$, although SResolve, which runs after the forward pass, will also use mutable arrays as in Section 9.1.) The given implementation of $\mathcal{M}$ has the atomic mutable cell in a reader context, and is a state monad in 'JobDescr': a description of a job that contains the *history* of a job together with its job ID and the number of sequential IDs generated in that job (numbered $0, \ldots, n-1$). The history of a job α is the subgraph of the job graph given by all jobs β satisfying $\beta < \alpha$ in the smallest *task* (recall Figure 12) containing the job. For the special case of the (unique) last job of a task, its history is precisely the whole task excluding itself. This definition of a "history" ensures that ($\star'$) has precisely the parts of the job graph that it needs to build up the job graph of the task that it itself is running in, which makes everything compose.

Differentiation. We keep the differentiation rules for all existing language constructs the same, except for changing the type of IDs to CID and using the monad $\mathcal{M}$ instead of doing manual state passing of the next ID to generate. A representative rule showing how this looks is: (compare Figure 8)

$$\mathbf{D}^5_c[(s,t)] = \mathbf{D}^5_c[s] \gg\!= \lambda x.\ \mathbf{D}^5_c[t] \gg\!= \lambda y.\ \textbf{return}\ (x, y)$$

https://doi.org/10.1017/S0956796825100051 Published online by Cambridge University Press

Because we now generate fresh compound IDs rather than plain integer IDs when executing primitive operations, we change $\mathbb{Z}$ to CID in $\mathbf{D}_c^4[\mathbb{R}]$:[41]

$$\mathbf{D}_c^5[\mathbb{R}] = (\mathbb{R}, (\mathrm{CID}, \underline{\mathbb{R}} \multimap (\mathrm{Staged}\, c \to \mathtt{IO}\ ())))$$

The new rule for the parallel pairing construct is simply:

$$\mathbf{D}_c^5[t \star s] = \mathbf{D}_c^5[t] \star' \mathbf{D}_c^5[s]$$

This is the only place where we use the operation $(\star')$, and thus the only place in the forward pass where we directly use the extended functionality of our monad $\mathcal{M}$.

12.4 Updating the wrapper

The wrapper is there to glue the various components together and to provide the (now parallel) definition of SResolve. We discuss the main ideas behind the parallel wrapper implementation in this section, skipping over some implementation details.

In Section 9, the backpropagators staged their backpropagator calls in a single array, indexed by their ID. With compound IDs, we still need one array slot for each backpropagator, meaning that we need a *nested* staging array:

$$\mathrm{Staged}\, c = (\mathrm{Array}\, \underline{\mathbb{R}}, \mathrm{Array}\, (\mathrm{Array}\, (\underline{\mathbb{R}} \multimap (\mathrm{Staged}\, c \to \mathtt{IO}\ ()), \underline{\mathbb{R}})))$$

where the outer array is indexed by the job ID and the inner by the sequential ID within that job. Because jobs have differing lengths, the nested arrays also have differing lengths and we cannot use a rectangular two-dimensional array. The implementation of SOneHot must be changed to update the cotangent collection array atomically because backpropagators, and hence SOneHot, will now be called from multiple threads. SCall needs a simple modification to (atomically) modify the correct element in the now-nested staging array.

To construct the initial Staged object, SAlloc needs to know the correct length of all the nested staging arrays; it can get this information from the job graph (in the History structure) that Wrap receives from the monadic evaluation of the forward pass.

This leaves the parallel implementation of SResolve. The idea here is to have two functions: one that resolves a *task* (this one is new and handles all parallelism) and one that resolves a *job* (and is essentially identical to the last version of SResolve, from Figure 10). A possible implementation is given in Figure 15, where the first function is called *resolveTask* and the second *resolveJob*. In this way, SResolve traverses the job graph (Figure 13(b)) from the terminal job backwards to the initial job, doing the usual sequential resolving process for each sequential job in the graph.

Duality. The usual mantra in reverse-mode AD is that "sharing in the primal becomes addition in the dual". When parallelism comes into play, not only do we have this duality in the data flow graph, we also get an interesting duality in the control-flow graph: SResolve forks where the primal joined and joins where the primal forked. Perhaps we can add a mantra for task-parallel reverse AD: "forks in the primal become joins in the dual".

[41] Here we work from the transformation $\mathbf{D}^4$ as described in Section 9.1 on monadic mutable arrays. With the defunctionalisation described in Section 10.2, the backpropagator would simply read 'Contrib' instead.

https://doi.org/10.1017/S0956796825100051 Published online by Cambridge University Press

```
SResolve : JobDescr → Staged c → IO (IArray ℝ)
SResolve jd (c, m) = resolveTask jd
  where resolveTask (history, jid, i) = do
          jobarr ← get jid m
          resolveJob (i − 1) jobarr
          case history of Start → return ()
                          Fork jd0 jd1 jd2 → do
                            inParallel (resolveTask jd1) (resolveTask jd2)
                            resolveTask jd0
        resolveJob (−1) arr = return ()
        resolveJob i arr = do
          (f, a) ← get i arr
          f a (c, m)
          resolveJob (i − 1) arr
```

Fig. 15. Implementation of SResolve for the parallel-ready dual-numbers reverse AD algorithm. The `inParallel` function is as in Figure 14.

13 Implementation

To show the practicality of our method, we provide a prototype implementation[42] of the parallel algorithm of Section 12, together with the improvements from Sections 10.1 and 10.2, that differentiates a sizeable fragment of Haskell98 including recursive types (reinterpreted as a strict, call-by-value language) using Template Haskell. As described in Section 9.1, we realise the mutable arrays using `IOVectors`. The implementation does not incorporate the changes given in Section 10.3 that transform the algorithm into classical taping (because implementations of taping already abound), but it does include support for recursive functions, coproduct types, and user-defined data types as described in Section 11.

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 Haskell `ad` library (Kmett and Contributors, 2021) (version 4.5.6) on a few basic functions.[43] These functions are the following (abbreviating `Double` as "`D`"):

- A single scalar multiplication of type `(D, D) -> D`;

[42] The code is available at https://github.com/tomsmeding/ad-dualrev-th.

[43] We use `Numeric.AD.Double` instead of `Numeric.AD` to allow `ad` to specialise for `Double`, which we also do. We keep `ad`'s (non-default) `ffi` flag off for a fairer playing ground (we could do similar things, but do not).

https://doi.org/10.1017/S0956796825100051 Published online by Cambridge University Press

Table 1. Benchmark results of Section 12 + Sections 10.1 and 10.2 versus `ad-4.5.6`. 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% bootstrapping confidence interval, rounded to 2 decimal digits. Setup: GHC 9.6.6 on Linux, Intel i9-10900K CPU, with Intel Turbo Boost disabled (i.e. running at a consistent 3.7 GHz).

	TH	`ad`	TH / `ad`
scalar mult.	0.33 μs ±0.00	0.80 μs ±0.00	≈0.4
dot product	0.95 μs ±0.03	2.14 μs ±0.07	≈0.4
sum-mat-vec	0.59 μs ±0.02	1.23 μs ±0.03	≈0.5
rotate_vec_by_quat	5.62 μs ±0.00	6.71 μs ±0.01	≈0.8
neural	2.4 ms ±0.05	8.1 ms ±0.01	≈0.3
particles (1 thr.)	7.6 ms ±0.05	9.1 ms ±0.08	≈0.8
particles (2 thr.)	4.6 ms ±0.04	—	—
particles (4 thr.)	2.4 ms ±0.12	—	—

- Dot product of type `([D], [D]) -> D`;
- Matrix–vector multiplication, then sum: of type `([[D]], [D]) -> D`;
- The `rotate_vec_by_quat` example from Krawiec et al. (2022) of type `(Vec3 D, Quaternion D) -> Vec3 D`, with `data Vec3 s = Vec3 s s s` and `data Quaternion s = Quaternion s s s s`;
- A simple, dense neural network with 2 hidden layers, ReLU activations and (safe) softmax output processing. The result vector is summed to make the output a single scalar. The actual Haskell function is generic in the number of layers and has type `([([[D]], [D])], [D]) -> D`: the first list contains a matrix and a bias vector for each hidden layer, and the second tuple component is the input vector. In the benchmark, the input has length 50, and the two hidden layers have size 100 and 50.
- Simulation of 4 particles in a simple force field with friction for 1000 time steps; this example has type `[((D, D), (D, D))] -> D`. The input is a list (of length 4) of initial positions and velocities; the output is $\sum_{(x,y)} x \cdot y$ ranging over the 4 result positions p, to ensure that the computation has a single scalar as output. The four particles are simulated in parallel using the ($\star$) combinator from Section 12.

The fourth test case, rotate_vec_by_quat, has a non-trivial return type; the benchmark executes its reverse pass three times ("3" being the number of scalars in the function output) to get the full Jacobian. The fifth test case, "particles", is run on 1, 2 and 4 threads, where the ideal result would be perfect scaling due to the four particles being independent.

The benchmark results are summarised in Figure 1 and shown in detail (including evidence of the desired linear complexity scaling) in Appendix A. The benchmarks are measured using the `criterion`[44] library. To get statistically significant results, we measure how the timings scale with increasing n:

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

https://doi.org/10.1017/S0956796825100051 Published online by Cambridge University Press

- Scalar multiplication, rotate_vec_by_quat, the neural network and the particle simulation 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.

The reported time is the deduced actual runtime for $n = 1$.

By the results in Figure 1, we see that on these simple benchmark programs, our implementation is faster than the existing `ad` library. While this is encouraging, it is not overly surprising: because our algorithm is implemented as a compile-time code transformation, the compiler is able to optimise the derivative code somewhat before it gets executed.

Of course, performance results may well be different for other programs, and AD implementations that have native support for array operations can handle some of these programs much more efficiently. Furthermore, there are various implementation choices for the code transformation that may seem relatively innocuous but have a large impact on performance (we give some more details below in Section 13.1).

For these reasons, 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). Nevertheless, our benchmarks include key components of many AD applications, and seeing that we have not at all special-cased their implementation (the implementation knows only primitive arithmetic operations such as scalar addition, multiplication, etc.), we believe that they suffice to demonstrate our limited claim.

13.1 Considerations for implementation performance

The target language of the code transformation in our implementation is Haskell, which is a lazy, garbage-collected language. This has various effects on the performance characteristics of our implementation.

Garbage collection. The graph of backpropagators (a normal data structure, "Contrib", since Section 10.2) is a big data structure of size proportional to the number of scalar operations performed in the source program. While this data structure grows during the forward pass, the nursery (zeroth generation) of GHC's generational garbage collector (GC) repeatedly fills up, triggering garbage collection of the heap. Because a GC pass takes time on the order of the amount of live data on the heap, these passes end up very expensive: a naive taping reverse AD algorithm becomes *quadratic* in the source program runtime. Using a GHC runtime system flag (e.g. `-A500m`) to set the nursery size of GHC's GC sufficiently large to prevent the GC from running during a benchmark (`criterion` runs the GC explicitly between each benchmark invocation), timings on some benchmarks above decrease significantly: on the largest benchmark (particles), this can save 10% off `ad`'s runtime and 25% off our runtime (though precise timings vary). The times reported in Figure 1 are with GHC's default GC settings.

While this is technically a complexity problem in our implementation, we gloss over this because it is not fundamental to the algorithm: the backpropagator graph does not contain

https://doi.org/10.1017/S0956796825100051 Published online by Cambridge University Press

cycles, so it could be tracked with reference-counting GC to immediately eliminate the quadratic blowup. Using a custom, manual allocator, one could also eliminate all tracking of the liveness of the graph because we know from the structure of the algorithm exactly when we can free a node in the graph (namely when we have visited it in the reverse pass). Our reference implementation does not do these things to be able to focus on the workings of the algorithm.

Laziness. Because data types are lazy by default in Haskell, a naive encoding of the Contrib data type from Section 10.2 would make the whole graph lazily evaluated (because it is never demanded during the forward pass). This results in a significant constant-factor overhead (more than $2\times$), and furthermore means that part of the work of the forward pass happens when the reverse pass first touches the top-level backpropagator; this work then happens sequentially, even if the forward pass was meant to be parallel. To achieve proper parallel scaling, it was necessary to make the Contrib graph strict, and furthermore to make the $\mathbf{D}_c[\mathbb{R}] = (\mathbb{R}, (\text{CID}, \text{Contrib}))$ triples strict, to ensure that the graph is fully evaluated as it is being *constructed*, not when it is demanded in the reverse pass.

Using some well-chosen `{-# UNPACK #-}` pragmas on some of these fields also had a significant positive effect on performance.

Thread pool. In Section 12, we used an abstract `inParallel` operation for running two jobs in parallel, assuming some underlying thread pool for efficient evaluation of the resulting parallel job graph. In a standard thread pool implementation, spawning tasks from within tasks can result in deadlocks. Because nested tasks are essential to our model of task parallelism, the implementer has to take care to further augment $\mathcal{M}$ from Section 12 to be a continuation monad as well: this allows the continuation of the `inParallel` operation to be captured and scheduled separately in the thread pool, so that thread pool jobs are indeed individual *jobs* as defined in Section 12.

The GHC runtime system has a thread scheduler that should handle this completely transparently, but in our tests it was not eager enough in assigning virtual Haskell threads to actual separate kernel threads, resulting in a sequential benchmark. This motivated a (small) custom thread pool implementation that sufficed for our benchmarks, but has a significant amount of overhead that can be optimised with more engineering effort.

Imperfect scaling. Despite efforts to the contrary, it is evident from Figure 1 that even on an embarrassingly parallel task like "particles", the implementation does not scale perfectly. From inspection of more granular timing of our implementation, we suspect that this is a combination of thread pool overhead and the fact that the forward pass simply allocates too quickly, exhausting the memory bandwidth of our system when run sufficiently parallel. However, more research is needed here to uncover the true bottlenecks.

14 Conclusions

One may ask: if the final algorithm from Section 9 can be argued to be "just taping" (Section 10.3), which is already widely used in practice—and the parallel extension is *still* just taping, except on a non-linear tape—what was the point? The point is the observation

https://doi.org/10.1017/S0956796825100051 Published online by Cambridge University Press

that optimisations are key to implementing efficient AD and that multiple kinds of reverse AD algorithms proposed by the programming languages community (in particular the one from Figure 6, studied by Brunel et al. (2020) and Huot et al. (2020, Section 6)—for further examples, see below in Section 15.2) tend to all reduce to taping after optimisation. We hope to have demonstrated that these techniques are quite flexible, allowing the differentiation of rich source languages with dynamic control flow, recursion and parallelism, and that the resulting algorithm can be relatively straightforward by starting from a naive differentiation algorithm and next optimising it to achieve the desired complexity.

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 are, for example, also related to the optimisations that fix the time complexity of CHAD (Smeding and Vákár, 2024)): sparse vectors are probably 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.

If one desires to take the techniques in this paper further to an algorithm that is useful in practice, it will be necessary to add *arrays* of scalars as a primitive type in the transformation. This would allow us to significantly reduce the size of the allocated tape and reuse much more of the structure of the original program in the reverse pass. Since many AD applications tend to involve very large arrays of scalars, we expect to be able to gain significant constant factor performance by replacing an array of many scalar backpropagators with a single vector backpropagator. We are currently exploring this direction of research.

15 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 Wengert (1964); Linnainmaa (1970); Speelpenning (1980). Good surveys can be found in Baydin et al. (2017) and 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 in Section 10.

15.1 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 in (Huot et al., 2020, Section 6) using denotational semantics. Brunel et al. (2020) introduce the key idea that underlies the optimisations in 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.

https://doi.org/10.1017/S0956796825100051 Published online by Cambridge University Press

Mazza and Pagani (2021) extend 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, Lucatelli Nunes and Vákár (2024) 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 Figure 6 applied to, essentially, idealised Haskell98.

15.2 Vectorised forward AD

There are strong parallels between our optimisations to the sequential algorithm in Sections 5 to 9 and the derivation by 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 $\mathbf{D}^1_c[\mathbb{R}] = (\mathbb{R}, \underline{\mathbb{R}} \multimap c)$, 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 $\underline{\mathbb{R}} \multimap c$ 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 (and indeed we do so, albeit later in Section 10.2).

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 6) 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 $\underline{\mathbb{R}} \multimap c \cong c$, the type Staged c can be thought of as a type Expr c of ASTs of expressions (with sharing) of type c.[45] The linear factoring rule

[45] In Krawiec et al. (2022), Expr c is called Delta.

https://doi.org/10.1017/S0956796825100051 Published online by Cambridge University Press

$f\ x + f\ y \rightsquigarrow f\ (x + y)$ for a linear function $f : \underline{\mathbb{R}} \multimap c$ 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), who try 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.

15.3 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: $\mathcal{R}_2$ from Section 4. This is the approach taken by 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) and Vytiniotis et al. (2019) and to sum and (co)inductive types in Nunes and Vákár (2023). Smeding and Vákár (2024) give a mechanised proof that the resulting algorithm attains the correct computational complexity, after a series of optimisations that are similar to the ones considered in this paper. 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.

Taping-like methods and non-local control flow. Another family of approaches to AD recently taken by the PL community contains those that rely on forms of non-local control flow such as delimited continuations (Wang et al., 2019) or effect handlers (Sigal, 2021; de Vilhena and Pottier, 2023). 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 passing around continuations, possibly combined with smart staging of those continuations like in this paper, this inversion of control can be deferred to run time by clever use of delimited control operators or effect handlers. De Vilhena and Pottier (2023) give a mechanised proof that the resulting (rather subtle) code transformation is correct.

Operationally, however, these effect-handler-based techniques are essentially equivalent to taping: the mutable cells for cotangent accumulation scope over the full remainder of the program. In this sense, they are operationally similar to the algorithm of Section 10.3,

https://doi.org/10.1017/S0956796825100051 Published online by Cambridge University Press

which is also essentially taping. The AD algorithm in Dex (Paszke et al., 2021*a*) also achieves something like taping in a functional style, by conversion to an A-normal form.

AD of parallel code. Bischof et al. (1991); Bischof (1991) present some of the first work in parallel AD. Rather than starting with a source program (with potential dynamic control flow) that has (fork-join) parallelism like we do and mirroring that in the reverse pass of the algorithm, they focus on code without dynamic control flow and analyse the dependency structure of the reverse pass at compile time to automatically parallelise it. This approach is developed further by Bucker et al. (2002).

Building on the classic imperative AD tool Tapenade (Hascoët and Pascual, 2013), Hückelheim and Hascoët (2022) discuss a method for differentiating parallel for-loops (with shared memory) in a parallelism-preserving way. Industrial machine learning frameworks such as TensorFlow (Abadi et al., 2016), JAX (Bradbury et al., 2018) and PyTorch (Paszke et al., 2017) focus on data parallelism through parallel array operations—typically first-order ones.

Kaler et al. (2021) focus on AD of programs with similar forms of fork-join parallelism as we consider in this paper. Their implementation builds on the Adept C++ library, which implements an AD algorithm that is very different from the one discussed in this paper. A unique feature of their work is that they give a formal analysis of the complexity of the technique and give bounds for both the work and the span of the resulting derivative code; it is possible that ideas from the cited work can be used to improve and bound the span of our parallel algorithm.

Other recent work has focussed on developing data-parallel derivatives for data-parallel array programs (Paszke et al., 2021*a*,*b*; Schenck et al., 2022). These methods are orthogonal to the ideas focussed on task parallelism that we present in the present paper.

In recent work, we have shown that an optimised version of CHAD seems to preserve task and data parallelism (Smeding and Vákár, 2024). We are working on giving formal guarantees about the extent (work and span) to which different forms of parallelism is preserved. Compared to the dual numbers reverse AD method of the present paper, CHAD seems to accommodate higher order array combinators more easily, including their data parallel versions. This makes us hopeful for the use of CHAD as a candidate algorithm for parallel AD.

Funding

We gratefully acknowledge funding for this research from NWO Veni grant number VI.Veni.202.124 and the ERC project FoRECAST.

Conflicts of interest

The authors report no conflict of interest.

https://doi.org/10.1017/S0956796825100051 Published online by Cambridge University Press

References

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., Levenberg, J., Monga, R., Moore, S., Murray, D. G., Steiner, B., Tucker, P. A., Vasudevan, V., Warden, P., Wicke, M., Yu, Y. & Zheng, X. (2016) Tensorflow: A system for large-scale machine learning. In 12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2–4, 2016. USENIX Association, pp. 265–283. Available at: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi

Abadi, M. & Plotkin, G. D. (2020) A simple differentiable programming language. *Proc. ACM Program. Lang.* **4**(POPL), 38:1–38:28.

Baydin, A. G., Pearlmutter, B. A., Radul, A. A. & Siskind, J. M. (2017) Automatic differentiation in machine learning: A survey. *J. Mach. Learn. Res.* **18**, 153:1–153:43.

Bischof, C. (1991) *Issues in Parallel Automatic Differentiation*. Lemont, IL: Argonne National Lab. Available at: https://ftp.mcs.anl.gov/pub/tech_reports/reports/P235.pdf

Bischof, C., Griewank, A. & Juedes, D. (1991) Exploiting parallelism in automatic differentiation. In Proceedings of the 5th International Conference on Supercomputing, pp. 146–153.

Boisseau, G. & Gibbons, J. (2018) What you needa know about yoneda: Profunctor optics and the yoneda lemma (functional pearl). *Proc. ACM Program. Lang.* **2**(ICFP), 84:1–84:27.

Bradbury, J., Frostig, R., Hawkins, P., Johnson, M. J., Leary, C., Maclaurin, D., Necula, G., Paszke, A., VanderPlas, J., Wanderman-Milne, S. & Zhang, Q. (2018) JAX: Composable transformations of Python+NumPy programs. Available at: https://github.com/jax-ml/jax

Brunel, A., Mazza, D. & Pagani, M. (2020) Backpropagation in the simply typed lambda-calculus with linear negation. *Proc. ACM Program. Lang.* **4**(POPL), 64:1–64:27.

Bucker, H., Lang, B., Rasch, A., Bischof, C. H. & an Mey, D. (2002) Explicit loop scheduling in openmp for parallel automatic differentiation. In Proceedings 16th Annual International Symposium on High Performance Computing Systems and Applications. IEEE, pp. 121–126.

de Vilhena, P. E. & Pottier, F. (2023) Verifying an effect-handler-based define-by-run reverse-mode ad library. *Log. Methods Comput. Sci.* **19**(4), 5:1–5:51.

Elliott, C. (2018) The simple essence of automatic differentiation. *Proc. ACM Program. Lang.* **2**(ICFP), 70:1–70:29.

Griewank, A. & Walther, A. (2008) *Evaluating Derivatives - Principles and Techniques of Algorithmic Differentiation*, 2nd ed. Philadephia, PA: SIAM.

Hascoët, L., Naumann, U. & Pascual, V. (2005) "To be recorded" analysis in reverse-mode automatic differentiation. *Future Gener. Comput. Syst.* **21**(8), 1401–1417.

Hascoët, L. & Pascual, V. (2013) The Tapenade automatic differentiation tool: Principles, model, and specification. *ACM Trans. Math. Softw.* **39**(3), 20:1–20:43.

Hückelheim, J. & Hascoët, L. (2022) Source-to-source automatic differentiation of openmp parallel loops. *ACM Trans. Math. Softw.* **48**(1), 1–32.

Hückelheim, J., Menon, H., Moses, W. S., Christianson, B., Hovland, P. D. & Hascoët, L. (2023) Understanding automatic differentiation pitfalls. CoRR: abs/2305.07546.

Hughes, R. J. M. (1986) A novel representation of lists and its application to the function "reverse". *Inf. Process. Lett.* **22**(3), 141–144.

Huot, M., Staton, S. & Vákár, M. (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. Cham, Switzerland: Springer, pp. 319–338. Available at: https://link.springer.com/book/10.1007/978-3-030-45231-5

Jacobs, K., Devriese, D. & Timany, A. (2022) Purity of an ST monad: Full abstraction by semantically typed back-translation. *Proc. ACM Program. Lang.* **6**(OOPSLA1), 1–27.

Kaler, T., Schardl, T. B., Xie, B., Leiserson, C. E., Chen, J., Pareja, A. & Kollias, G. (2021) Parad: A work-efficient parallel algorithm for reverse-mode automatic differentiation. In Symposium on Algorithmic Principles of Computer Systems (APOCS). SIAM, pp. 144–158.

Kmett, E. & Contributors. (2021) *Ad: Automatic Differentiation*. Haskell Library on Hackage.

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

Launchbury, J. & Jones, S. L. P. (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. New York, NY: ACM, pp. 24–35. Available at: https://dl.acm.org/doi/proceedings/10.1145/178243

Linnainmaa, S. (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)*. University of Helsinki.

Lucatelli Nunes, F. & Vákár, M. (2024) Automatic differentiation for ML-family languages: Correctness via logical relations. *Math. Struct. Comput. Sci*. **34**(8), 747–806.

Margossian, C. C. (2019) A review of automatic differentiation and its efficient implementation. *Wiley Interdiscip. Rev. Data Min. Knowl. Discov.* **9**(4), e1305.

Mazza, D. & Pagani, M. (2021) Automatic differentiation in PCF. *Proc. ACM Program. Lang.* **5**(POPL), 1–27.

Nunes, F. L. & Vákár, M. (2023) CHAD for expressive total languages. *Math. Struct. Comput. Sci.* **33**(4-5), 311–426.

Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L. & Lerer, A. (2017) Automatic differentiation in PyTorch. In NIPS 2017 Autodiff Workshop: The Future of Gradient-Based Machine Learning Software and Techniques. Red Hook, NY, USA. Curran Associates, Inc. Available at: https://dl.acm.org/doi/proceedings/10.5555/3294996

Paszke, A., Johnson, D. D., Duvenaud, D., Vytiniotis, D., Radul, A., Johnson, M. J., Ragan-Kelley, J. & Maclaurin, D. (2021a) Getting to the point: Index sets and parallelism-preserving autodiff for pointful array programming. *Proc. ACM Program. Lang.* **5**(ICFP), 1–29.

Paszke, A., Johnson, M. J., Frostig, R. & Maclaurin, D. (2021b) Parallelism-preserving automatic differentiation for second-order array languages. In Proceedings of the 9th ACM SIGPLAN International Workshop on Functional High-Performance and Numerical Computing, pp. 13–23.

Pearlmutter, B. A. & Siskind, J. M. (2008) Reverse-mode AD in a functional framework: Lambda the ultimate backpropagator. *ACM Trans. Program. Lang. Syst.* **30**(2), 7:1–7:36.

Reynolds, J. C. (1998) Definitional interpreters for higher-order programming languages. *High. Order Symb. Comput.* **11**(4), 363–397.

Schenck, R., Rønning, O., Henriksen, T. & Oancea, C. E. (2022) AD for an array language with nested parallelism. In SC22: International Conference for High Performance Computing, Networking, Storage and Analysis, Dallas, TX, USA, November 13–18, 2022. IEEE, pp. 58:1–58:15.

Shaikhha, A., Fitzgibbon, A., Vytiniotis, D. & Jones, S. P. (2019) Efficient differentiable programming in a functional array-processing language. *Proc. ACM Program. Lang.* **3**(ICFP), 97:1–97:30.

Sheard, T. & Jones, S. L. P. (2002) Template meta-programming for Haskell. *ACM SIGPLAN Not*. **37**(12), 60–75.

Sigal, J. (2021) Automatic differentiation via effects and handlers: An implementation in frank. arXiv preprint: arXiv:2101.08095.

Siskind, J. M. & Pearlmutter, B. A. (2018) Divide-and-conquer checkpointing for arbitrary programs with no user annotation. *Optimiz. Meth. Softw*. **33**(4–6), 1288–1330.

Smeding, T. & Vákár, M. (2022) Efficient dual-numbers reverse AD via well-known program transformations. CoRR: abs/2207.03418v2.

Smeding, T. & Vákár, M. (2023) Efficient dual-numbers reverse AD via well-known program transformations. *Proc. ACM Program. Lang.* **7**(POPL), 1573–1600.

Smeding, T. & Vákár, M. (2024) Efficient CHAD. *Proc. ACM Program. Lang.* **8**(POPL), 1060–1088.

Speelpenning, B. (1980) *Compiling Fast Partial Derivatives of Functions Given by Algorithms*. Technical report. Illinois University.

https://doi.org/10.1017/S0956796825100051 Published online by Cambridge University Press

Vákár, M. (2021) Reverse AD at higher types: Pure, principled and denotationally correct.In Programming Languages and Systems. Springer, pp. 607–634.
Vákár, M. & Smeding, T. (2022) CHAD: Combinatory homomorphic automatic differentiation, pp. 20:1–20:49. Available at: https://dl.acm.org/doi/10.1145/3527634
Vytiniotis, D., Belov, D., Wei, R., Plotkin, G. & Abadi, M. (2019) The differentiable curry. In *NeurIPS Workshop on Program Transformations*.
Wang, F., Zheng, D., Decker, J. M., Wu, X., Essertel, G. M. & Rompf, T. (2019) Demystifying differentiable programming: Shift/reset the penultimate backpropagator. *Proc. ACM Program. Lang.* **3**(ICFP), 96:1–96:31.
Wengert, R. E. (1964) A simple automatic derivative evaluation program. *Commun. ACM*. **7**(8), 463–464.
Westrick, S., Fluet, M., Rainey, M. & Acar, U. A. (2024) Automatic parallelism management. *Proc. ACM Program. Lang.* **8**(POPL), 1118–1149.

Appendix A

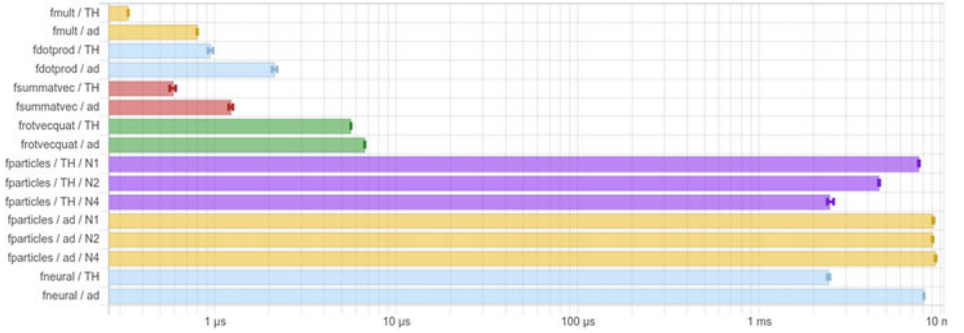
On the following page, we show benchmark details from our implementation on some sequential programs, as determined by the Haskell library `criterion`. See Section 13.

https://doi.org/10.1017/S0956796825100051 Published online by Cambridge University Press

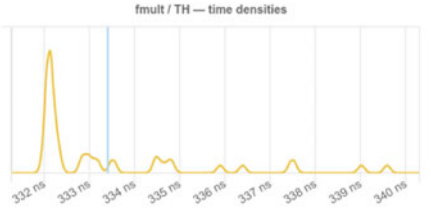
A Criterion Benchmark Details

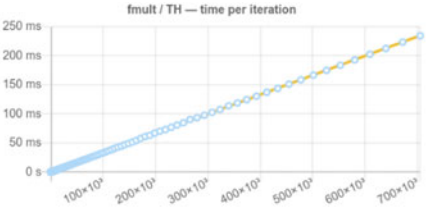
criterion performance measurements

overview



fmult / TH

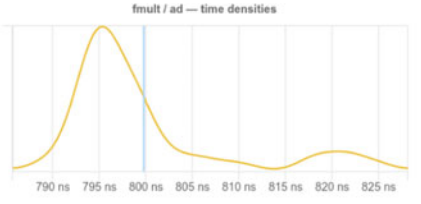


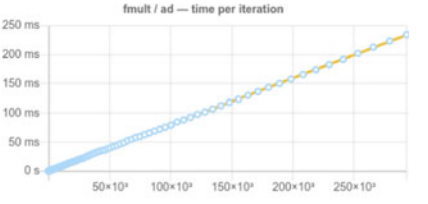


	lower bound	**estimate**	upper bound
OLS regression	332 ns	**333 ns**	333 ns
R² goodness-of-fit	1.00	**1.00**	1.00
Mean execution time	333 ns	**333 ns**	334 ns
Standard deviation	1.43 ns	**1.96 ns**	2.61 ns

Outlying measurements have no (0.426%) effect on estimated standard deviation.

fmult / ad



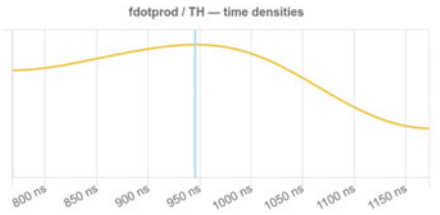


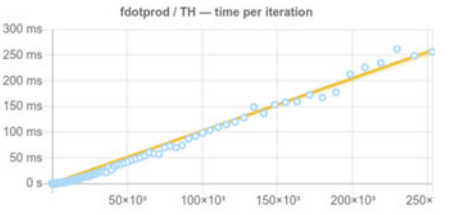
	lower bound	**estimate**	upper bound
OLS regression	796 ns	**797 ns**	798 ns
R² goodness-of-fit	1.00	**1.00**	1.00
Mean execution time	798 ns	**800 ns**	803 ns
Standard deviation	6.35 ns	**8.69 ns**	11.1 ns

Outlying measurements have a slight (8.32%) effect on estimated standard deviation.

https://doi.org/10.1017/S0956796825100051 Published online by Cambridge University Press

fdotprod / TH

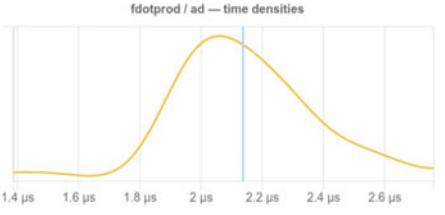


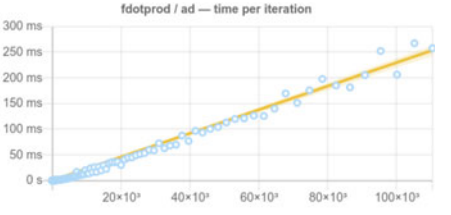


	lower bound	**estimate**	upper bound
OLS regression	991 ns	**1.02 µs**	1.05 µs
R² goodness-of-fit	0.988	**0.991**	0.994
Mean execution time	918 ns	**945 ns**	978 ns
Standard deviation	80.0 ns	**91.7 ns**	110 ns

Outlying measurements have a severe (88.6%) effect on estimated standard deviation.

fdotprod / ad

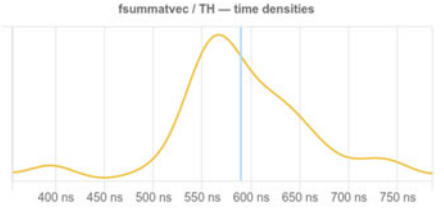


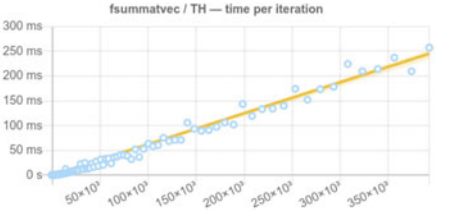


	lower bound	**estimate**	upper bound
OLS regression	2.21 µs	**2.30 µs**	2.39 µs
R² goodness-of-fit	0.985	**0.989**	0.993
Mean execution time	2.07 µs	**2.14 µs**	2.21 µs
Standard deviation	168 ns	**213 ns**	288 ns

Outlying measurements have a severe (88.1%) effect on estimated standard deviation.

fsummatvec / TH



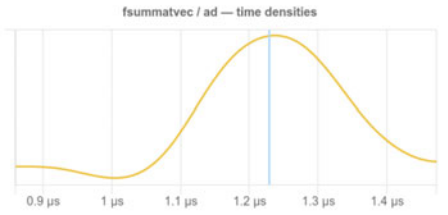


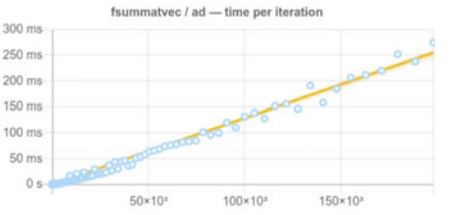
	lower bound	**estimate**	upper bound
OLS regression	599 ns	**624 ns**	646 ns
R² goodness-of-fit	0.982	**0.988**	0.992
Mean execution time	564 ns	**589 ns**	609 ns
Standard deviation	54.4 ns	**71.1 ns**	99.3 ns

Outlying measurements have a severe (92.4%) effect on estimated standard deviation.

https://doi.org/10.1017/S0956796825100051 Published online by Cambridge University Press

fsummatvec / ad

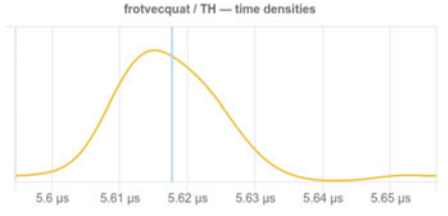


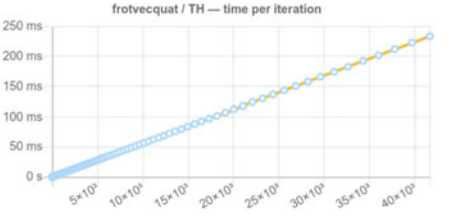


	lower bound	estimate	upper bound
OLS regression	1.24 µs	1.29 µs	1.32 µs
R² goodness-of-fit	0.987	0.991	0.994
Mean execution time	1.19 µs	1.23 µs	1.26 µs
Standard deviation	77.0 ns	106 ns	149 ns

Outlying measurements have a severe (85.3%) effect on estimated standard deviation.

frotvecquat / TH

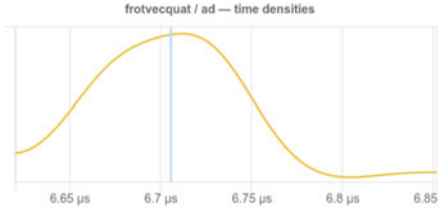


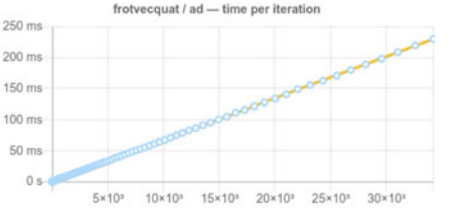


	lower bound	estimate	upper bound
OLS regression	5.61 µs	5.62 µs	5.62 µs
R² goodness-of-fit	1.00	1.00	1.00
Mean execution time	5.62 µs	5.62 µs	5.62 µs
Standard deviation	5.89 ns	8.24 ns	13.1 ns

Outlying measurements have no (0.565%) effect on estimated standard deviation.

frotvecquat / ad



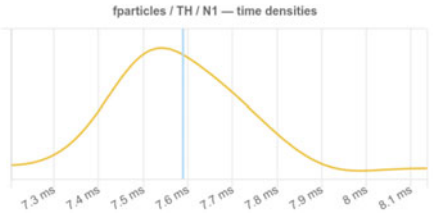


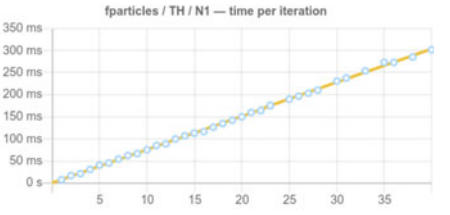
	lower bound	estimate	upper bound
OLS regression	6.69 µs	6.70 µs	6.71 µs
R² goodness-of-fit	1.00	1.00	1.00
Mean execution time	6.70 µs	6.71 µs	6.72 µs
Standard deviation	28.6 ns	36.3 ns	56.0 ns

Outlying measurements have no (0.578%) effect on estimated standard deviation.

https://doi.org/10.1017/S0956796825100051 Published online by Cambridge University Press

fparticles / TH / N1

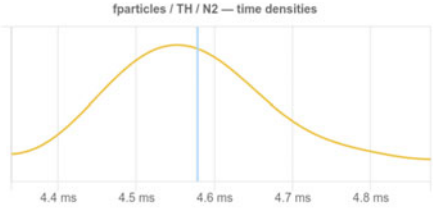


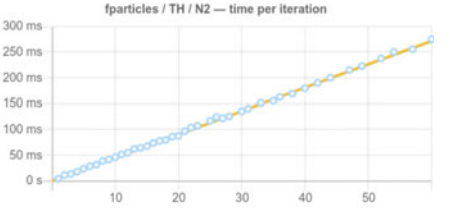


	lower bound	estimate	upper bound
OLS regression	7.51 ms	7.59 ms	7.68 ms
R^2 goodness-of-fit	0.999	0.999	1.00
Mean execution time	7.54 ms	7.59 ms	7.64 ms
Standard deviation	114 µs	149 µs	231 µs

Outlying measurements have a slight (2.85%) effect on estimated standard deviation.

fparticles / TH / N2

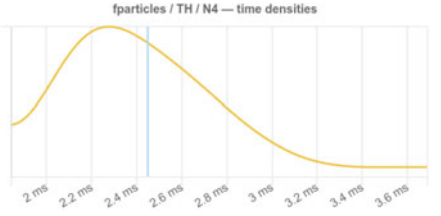


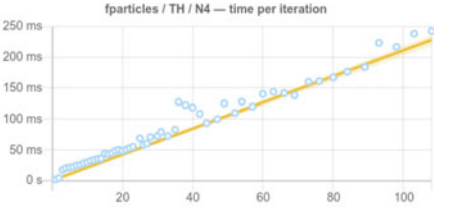


	lower bound	estimate	upper bound
OLS regression	4.48 ms	4.53 ms	4.57 ms
R^2 goodness-of-fit	0.999	0.999	0.999
Mean execution time	4.55 ms	4.58 ms	4.61 ms
Standard deviation	81.9 µs	104 µs	133 µs

Outlying measurements have a slight (8.87%) effect on estimated standard deviation.

fparticles / TH / N4



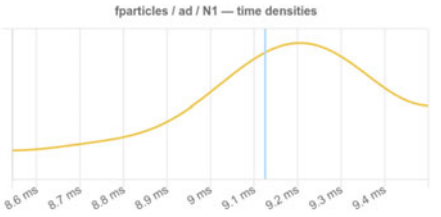


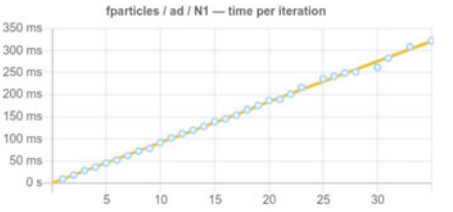
	lower bound	estimate	upper bound
OLS regression	2.03 ms	2.11 ms	2.19 ms
R^2 goodness-of-fit	0.952	0.973	0.991
Mean execution time	2.37 ms	2.45 ms	2.57 ms
Standard deviation	267 µs	330 µs	445 µs

Outlying measurements have a severe (78.6%) effect on estimated standard deviation.

https://doi.org/10.1017/S0956796825100051 Published online by Cambridge University Press

fparticles / ad / N1

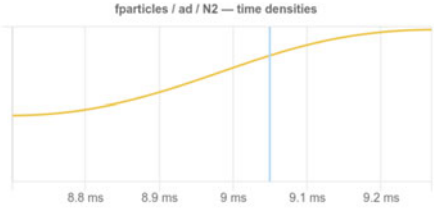


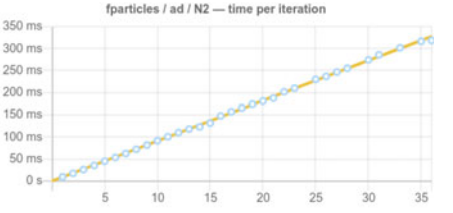


	lower bound	**estimate**	upper bound
OLS regression	9.02 ms	**9.18 ms**	9.33 ms
R² goodness-of-fit	0.997	**0.998**	1.00
Mean execution time	9.04 ms	**9.12 ms**	9.19 ms
Standard deviation	162 µs	**207 µs**	269 µs

Outlying measurements have a slight (5.17%) effect on estimated standard deviation.

fparticles / ad / N2

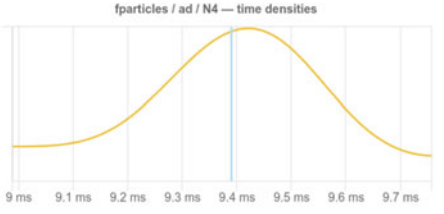


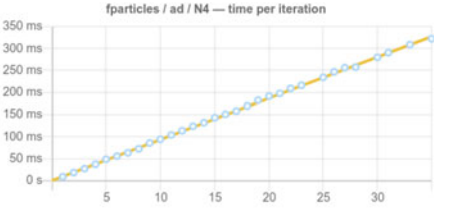


	lower bound	**estimate**	upper bound
OLS regression	8.98 ms	**9.09 ms**	9.20 ms
R² goodness-of-fit	0.999	**0.999**	1.00
Mean execution time	9.00 ms	**9.05 ms**	9.10 ms
Standard deviation	113 µs	**143 µs**	179 µs

Outlying measurements have a slight (3.03%) effect on estimated standard deviation.

fparticles / ad / N4



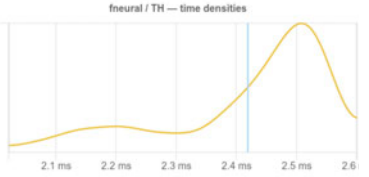


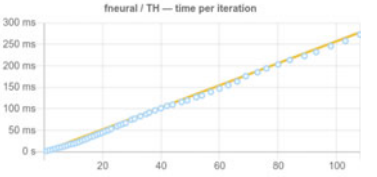
	lower bound	**estimate**	upper bound
OLS regression	9.24 ms	**9.33 ms**	9.44 ms
R² goodness-of-fit	0.999	**0.999**	1.00
Mean execution time	9.33 ms	**9.39 ms**	9.44 ms
Standard deviation	114 µs	**149 µs**	196 µs

Outlying measurements have a slight (3.12%) effect on estimated standard deviation.

https://doi.org/10.1017/S0956796825100051 Published online by Cambridge University Press

fneural / TH

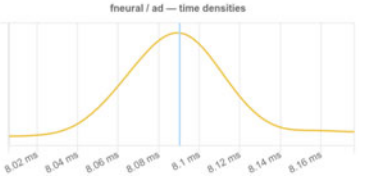


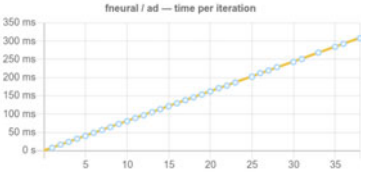


	lower bound	**estimate**	upper bound
OLS regression	2.55 ms	**2.57 ms**	2.59 ms
R^2 goodness-of-fit	0.999	**0.999**	0.999
Mean execution time	2.37 ms	**2.42 ms**	2.46 ms
Standard deviation	104 µs	**134 µs**	169 µs

Outlying measurements have a moderate (39.2%) effect on estimated standard deviation.

fneural / ad





	lower bound	**estimate**	upper bound
OLS regression	8.11 ms	**8.12 ms**	8.14 ms
R^2 goodness-of-fit	1.00	**1.00**	1.00
Mean execution time	8.08 ms	**8.09 ms**	8.10 ms
Standard deviation	20.9 µs	**28.6 µs**	39.3 µs

Outlying measurements have a slight (2.94%) effect on estimated standard deviation.

understanding this report

In this report, each function benchmarked by criterion is assigned a section of its own.

- The chart on the left is a kernel density estimate (also known as a KDE) of time measurements. This graphs the probability of any given time measurement occurring. A spike indicates that a measurement of a particular time occurred; its height indicates how often that measurement was repeated.
- The chart on the right is the raw data from which the kernel density estimate is built. The *x*-axis indicates the number of loop iterations, while the *y*-axis shows measured execution time for the given number of loop iterations. The line behind the values is the linear regression estimate of execution time for a given number of iterations. Ideally, all measurements will be on (or very near) this line. The transparent area behind it shows the confidence interval for the execution time estimate.

Under the charts is a small table. The first two rows are the results of a linear regression run on the measurements displayed in the right-hand chart.

- *OLS regression* indicates the time estimated for a single loop iteration using an ordinary least-squares regression model. This number is more accurate than the *mean* estimate below it, as it more effectively eliminates measurement overhead and other constant factors.
- *R^2; goodness-of-fit* is a measure of how accurately the linear regression model fits the observed measurements. If the measurements are not too noisy, R^2; should lie between 0.99 and 1, indicating an excellent fit. If the number is below 0.99, something is confounding the accuracy of the linear model.
- *Mean execution time* and *standard deviation* are statistics calculated from execution time divided by number of iterations.

We use a statistical technique called the bootstrap to provide confidence intervals on our estimates. The bootstrap-derived upper and lower bounds on estimates let you see how accurate we believe those estimates to be.

A noisy benchmarking environment can cause some or many measurements to fall far from the mean. These outlying measurements can have a significant inflationary effect on the estimate of the standard deviation. We calculate and display an estimate of the extent to which the standard deviation has been inflated by outliers.

colophon

This report was created using the criterion benchmark execution and performance analysis tool.

Criterion is developed and maintained by Bryan O'Sullivan.

https://doi.org/10.1017/S0956796825100051 Published online by Cambridge University Press