Reverse-mode Automatic Differentiation for Functional Array Languages

Tom Smeding

Reverse-mode Automatic Differentiation for Functional Array Languages

Automatisch Differentiëren in Omgekeerde Modus voor Functionele Arrayprogrammeertalen

(met een samenvatting in het Nederlands)

Proefschrift

ter verkrijging van de graad van doctor aan de
Universiteit Utrecht
op gezag van de
rector magnificus, prof. dr. ir. W. Hazeleger,
ingevolge het besluit van het College voor Promoties
in het openbaar te verdedigen op

door

Tom Jacob Smeding

geboren op 21 februari 1998
te Den Haag

Promotor:
Prof. dr. Gabriele K. Keller

Copromotor:
Dr. Matthijs I.L. Vákár

Beoordelingscommissie:

Prof. dr. Fritz Henglein	University of Copenhagen, Denmark
Prof. dr. Johan T. Jeuring	Utrecht University
Prof. dr. Barak A. Pearlmutter	Maynooth University, Ireland
Prof. dr. Sven-Bodo Scholz	Radboud University, The Netherlands
Dr. Dimitrios Vytiniotis	DeepMind, United Kingdom

The work in the thesis has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics).

Abstract

Purely functional array programming languages, especially second-order ones, are a high-level tool for expressing parallel numerical computations; their explicit structure means that compilers can optimise them very effectively. If such numerical computations require derivatives of functions that are complicated or that are subject to changing requirements, the algorithm of choice for computing those derivatives is usually *automatic differentiation* (AD), and particularly *reverse-mode AD*. However, existing fast implementations of reverse-mode AD are generally written for first-order languages — imperative and functional. It turns out to be difficult to design a reverse-mode AD algorithm that not only supports parallel input languages with second-order array operations, but that furthermore has the right complexity and is efficient in practice.

In this thesis, we make progress on this topic by considering two candidate reverse-mode AD algorithms for parallel functional array languages: *dual-numbers reverse AD* and *Combinatory Homomorphic Automatic Differentiation* (CHAD). For both algorithms, we discuss theory, intuition, operational characteristics and how to extend them to higher-order parallel functional languages, and we provide perspectives on practical performance. An additional goal of this thesis is to provide a thorough discussion of common issues in reverse-mode AD for a functional programming audience.

As expected, we find an inherent tension between efficiency, simplicity and generalisability of reverse AD algorithms. It seems that dual-numbers reverse AD can be either simple and generalisable, or efficient — not both at the same time. CHAD shows promise for a better trade-off, but more research is needed, especially on the topic of practical efficiency.

Contents

1 Introduction

Computing hardware is getting more and more parallel. The past roughly 20 years have brought us many-core CPUs, as well as general-purpose programmable GPUs that offer much, much wider parallelism — but also on a smaller scale, even within a single, ostensibly sequential CPU core, we have not only explicit parallelism in the form of SIMD instructions, but also implicit *instruction-level parallelism* exploited live by the hardware itself using clever scheduling and redundant silicon.

Programming tutorials and introductory classes typically ignore these developments completely, however. Parallel programming is seen as difficult, and not without reason; if we naively extend traditional imperative programming to parallel computing environments, we suddenly have to deal with a whole class of problems not seen before. The ones that arise from dealing with concurrency tend to be the most famous, as they have been the source of many security bugs: small mistakes in synchronisation, communication or scheduling can lead to race conditions, deadlocks and other issues. But because we want to use all that parallel hardware effectively, we also have parallelism-related problems: how do we communicate between threads *efficiently*? How do we make the best use of caches or heterogeneous hardware, or even distribute work over parallel threads? Some of these difficulties can be addressed with clever libraries or type systems (e.g. of the Rust language), but writing effective parallel code in the imperative style remains a specialist's domain, especially programming for GPUs.

Functional programming offers a very different approach to programming with concurrency and parallelism. The lack of free mutation seems restrictive — and sometimes it is — but having no concurrency bugs by construction, and being promised clever compilers that figure out all the parallelism choices for you, is certainly tempting. *Functional array languages* try to make this promise a reality; overtly functional examples are languages such as Futhark [Henriksen 2017], Accelerate [Chakravarty et al. 2011], SaC [Scholz 2003], Lift [Steuwer et al. 2017], Halide [Ragan-Kelley et al. 2013] and of course APL [Iverson 1962], but even frameworks that market themselves as practical first and foremost, tend to be secretly functional: TensorFlow [Abadi et al. 2016], as well as XLA [OpenXLA

contributors 2022] (which is the backend for JAX [Bradbury et al. 2018], TensorFlow and optionally PyTorch [Paszke et al. 2017]), are functional array languages, and XLA is in fact not dissimilar from Accelerate. In Section 2.1, we give an overview of the kind of functional array languages we consider in this thesis with their most important properties.

Numerical optimisation. In the context of numerical computation, where one makes use of arrays extensively enough that special array languages can bring significant benefits, it is not uncommon to be implementing some kind of model (e.g. a regression model) that must be fitted to collected data. While there are optimisation (fitting) algorithms, such as evolutionary algorithms, that work for completely black-box loss functions[1], many problems have *some* kind of continuous structure. If such structure is present, the *derivative* of the loss function gives valuable local information about the "optimisation landscape", which can help guide an optimiser in the right direction. Optimisation algorithms that use derivatives include, naturally, gradient descent and its many variations, but also L-BFGS (e.g. [Liu and Nocedal 1989]), Newton optimisation (which uses second-order derivatives), and others.

Note that in this context, the function that we need a derivative of is implemented in code (possibly in such a functional array language): while an algebraic form that can be manually differentiated is sometimes also available (for example in the case of a classical, dense neural network), it would be quite convenient if the implementation itself could be differentiated directly. This saves work and prevents mistakes (think, for example, of manually differentiating a complicated recurrent neural network), and it frees up time for modelling the problem in the domain at hand.

Bayesian probabilistic inference. Outside of optimisation, another context where derivatives of functions implemented in code turn up, perhaps surprisingly for some readers, is Bayesian probabilistic inference. The premise here is that one has a probabilistic model, often expressed in a probabilistic programming language like Stan [Carpenter et al. 2017] or PyMC [Pla et al. 2023], that describes a *prior* belief on the parameters $\theta \in \Theta$ of the model (described by a probability density function $p(\theta)$), together with a conditional probability of observations (D, "data") given parameters (a density function $p(D \mid \theta)$). The desire is to compute $p(\theta \mid D)$: the *posterior* density function, describing an updated belief on the parameters given some observations. Bayes' formula reads as follows in this case:

$$p(\theta \mid D) = \frac{p(D \mid \theta)\, p(\theta)}{p(D)}$$

[1] Other names include: scoring function, distance function.

The denominator, $p(D)$, expands to $\int_\Theta p(D \mid \theta)\, p(\theta)\, d\theta$ for a continuous parameter space Θ. Because parameter spaces are high-dimensional and probabilistic models are non-trivial, this integral is in practice hopeless to compute algebraically or even numerically. However, because we are interested in $p(\theta \mid D)$ as a function of θ (i.e. for a fixed dataset of observations D), that denominator is a constant; hence the numerator, $p(D \mid \theta)\, p(\theta)$ as a function of θ, is already an *unnormalised* version of the density function of the posterior distribution.

Now, the trick is to not attempt to actually compute $p(\theta \mid D)$, but to merely *sample* from the posterior distribution. This is possible using the Metropolis–Hastings (M–H) algorithm [Metropolis et al. 1953; Hastings 1970], which builds a sampler for a probability distribution on Θ — say, with density $q(\theta)$ — based on almost any[2] given Markov chain in Θ. M–H is a very suitable algorithm because it computes only ratios $\frac{q(\theta_1)}{q(\theta_2)}$, and hence q is allowed to be unnormalised.

Naturally, the quality and efficiency of the resulting sampler depends heavily on the Markov chain in question, and it turns out that for continuous parameter spaces, the Hamiltonian Monte Carlo (HMC) algorithm [Neal 2011] does a remarkably good job. It instantiates M–H with a Markov chain formed by repeatedly simulating a short trajectory of a physical particle on a frictionless surface with height $h(\theta) = -\log(q(\theta))$, where q is the density of our target distribution. Raising this surface by a constant amount (i.e. multiplying q by a constant factor) does not change the behaviour of the particle, so like M–H itself, HMC does not depend on the normalisation of q; this means that it works on $p(D \mid \theta)\, p(\theta)$ directly, which we can simply compute from givens.

To compute the forces on the particle in HMC's physics simulation, the slope of the landscape (i.e. the gradient of $h(\theta)$) is required, which means that we must differentiate q, the density function given by the user. This way, an intractable integration problem was replaced by a (much easier) differentiation problem.

Automatic differentiation. Because we do not want to ask the user to manually differentiate any probabilistic model they wish to run inference on, or to manually differentiate any function they wish to numerically optimise, we want to automatically differentiate mathematical functions implemented as code. The class of algorithms doing this is unsurprisingly called *automatic differentiation*, also known as "autodiff"; in this thesis, we will abbreviate it to *AD*.

Most standard AD algorithms can be classified either as *forward-mode* or as *reverse-mode*; we will refer to these modes as *forward AD* and *reverse AD*, respectively. Roughly speaking, forward AD is efficient on functions of type $\mathbb{R} \to \mathbb{R}^n$, whereas reverse AD is efficient on functions of type $\mathbb{R}^n \to \mathbb{R}$. Forward AD is much easier to implement and get efficient than reverse AD, yet reverse AD

[2]The input Markov chain, the *proposal distribution*, must be irreducible.

is often the one that is most useful in practice: indeed, in both examples given above (numerical optimisation and Bayesian inference), the function in question has many[3] parameters, but its output (the objective for optimisation; the surface height for Hamiltonian Monte Carlo) is just a single real scalar. In Section 2.2, we give an in-depth explanation of AD, what it computes and why its two main modes are efficient on the two mentioned function signatures.

Since AD is a class of algorithms, there is a design space to explore; the impetus behind this thesis is to find a reverse AD algorithm that satisfies all of the following:

1. It should be simple enough that we can prove it correct. It is acceptable if this proceeds by informally showing it semantically equivalent to a different (even simpler) algorithm, which in turn has a more formal semantical correctness treatment.

2. It should compute gradients with the correct (optimal [Griewank and Walther 2008]) computational complexity: only a constant-factor overhead over the original function, which only computed the function result, not its derivative.

3. It should compute gradients fast in practice.

4. It should work well on second-order (see Section 2.1.1) functional array languages.

5. It can handle parallel input programs, and preserve that parallelism in the derivative: that is, the derivative computation corresponding to some parallel subcomputation should run in parallel to a similar extent.

6. Preferably, it also supports higher-order input programs, i.e. with first-class function values; functional array languages that support higher-order code are rare, and they are already very useful even without full support for function values. Hence, this point has lower priority.

All AD algorithms that we are aware of, so far, fail at least one of these requirements. (See Sections 2.2.8 to 2.2.10, as well as the literature overview sections in each of the chapters, for an overview of existing algorithms.) In this thesis, we approach the problem by starting from algorithms that are already firmly rooted in theory; we then make them efficient in practice, step by step. While we do not quite achieve the ideal algorithm, we provide a novel perspective on this corner of the design space, and perhaps we even get somewhat close.

[3]Sometimes, e.g. with extravagant large language models, even hundreds of billions.

1.1 Contributions

We discuss in detail in this thesis two algorithms for reverse AD: *dual-numbers reverse AD* and *Combinatory Homomorphic Automatic Differentiation* (CHAD).

- First, in Chapter 3, we introduce the dual-numbers reverse AD algorithm and optimise it to have the correct computational complexity. The presentation is based on [Smeding and Vákár 2023a], presented at POPL '23, and has a prototype implementation in Haskell. In Section 3.10, we extend this to task parallelism as described in [Smeding and Vákár 2025], an extended version of the POPL '23 article published in JFP.

- In Chapter 4, we present a different extension to the base dual-numbers reverse AD algorithm to efficiently differentiate bulk array operations, including support for one second-order operation: 'build' (also known as 'tabulate' or 'generate'). This is based on a preprint [Smeding et al. 2025] in collaboration with Mikołaj Konarski, Simon Peyton Jones and Andrew Fitzgibbon; the accompanying implementation is primarily by Mikołaj Konarski.

- Then, in Chapter 5, we start the second part of this thesis by introducing the CHAD algorithm from [Vákár and Smeding 2022] for a functional programming audience. The conception and semantical analysis using category theory and combinators are by Matthijs Vákár; the presentation in terms of the lambda calculus, the original prototype implementation, and the new exposition in this thesis, are by the author.

- We continue in Chapter 6 with improvements to the CHAD algorithm to achieve the correct complexity, culminating in a complexity proof formalised in Agda. We also provide perspectives on an extension to parallel functional array languages. This chapter is based on [Smeding and Vákár 2024], presented at POPL '24.

- Finally, in Chapter 7, we present a number of optimisations to the algorithm of Chapter 6 that benefit practical performance. We provide an implementation in Haskell that includes these optimisations as well as a number of others. This chapter is unpublished outside of this thesis and describes in-progress research by the author.

We include background and a literature overview in Chapter 2, including a detailed introduction to AD that builds useful intuition in Section 2.2. We end with conclusions and future work in Chapter 8.

Authorship and supervision. This dissertation is written by the author. Unless explicitly stated otherwise, technical development (including proofs and complexity analyses), implementations, and the initial drafting of the chapters were led by the author as part of his PhD. Chapter-specific authorship notes are given as an unnumbered footnote on the first page of the appropriate chapter. Dr. Matthijs Vákár acted as daily supervisor and co-author on several chapters; his contribution was primarily through supervision, setting the research direction, discussions and detailed feedback, while giving the author the independence to take full ownership of the work.

2 Background

This chapter functions both as an introduction to the concepts upon which we build, and as an overview of the most relevant literature on the topic. While the primary focus of this thesis is AD, the structure and performance characteristics of functional array languages do inform some algorithm design decisions in this thesis. Therefore, we introduce them to the extent that we need in Section 2.1, before we move on to AD in Section 2.2.

We would also like to point out to the reader that there is an index of terminology in the back of this book.

2.1 Functional Array Languages

A *functional array language* is a functional *domain-specific language* (DSL) for computations centered around arrays. Thus, such languages are designed around bulk operations on arrays: the idea is to perform a small number of operations on large amounts of data each time, instead of many operations on individual data elements. The motivation is performance: performing a single operation on a single data element has a certain amount of minimal runtime overhead. Of course, how much overhead this has depends on the operation to perform, the data representation and even the hardware that the program is run on, but certainly, one operation takes at least one machine instruction. A modern CPU can run billions of such instructions per second in a single instruction stream, but when compared with properly using all its parallelism capabilities, and especially compared with the computational capacity of a modern GPU, "a few billion" suddenly lacks three or four zeros.

Both task parallelism (using multiple pieces of hardware at the same time, each running a separate instruction stream) and data parallelism (executing the same operation on multiple data elements while paying the overhead for just one such operation) require more *structure* than simply a stream of individual operations. In this situation, the functional approach to language design is to provide general building blocks that are expressive enough to implement a wide variety

of algorithms, while having the required parallelism structure by construction. Functional array languages attempt precisely this.

Generally, useful functional array languages come in two styles: first-order and second-order. Because the distinction turns out to be important for AD, we discuss these styles in more detail in the following subsections.

2.1.1 Order

Because supporting full, unrestricted lambda abstraction is a significant additional burden for a language implementation (or a code transformation like AD) and furthermore not always beneficial for performance, not all programming languages support functions to the same extent. However, the terminology used to describe these restrictions on lambda abstraction varies somewhat across literature. The terminology we choose here generally corresponds to the convention in functional array languages, and is most suited to the AD work we present in this thesis.

- *First-order language*: a language with (n-ary) built-in functions; lambda abstraction is not available. Sometimes closed (top-level) user-defined function definitions *are* allowed. A *first-order array operation* in an array language is, for example, transpose : Matrix $a \to$ Matrix a; a negative example is map : $(a \to b) \to$ Array $a \to$ Array b, which is in fact a second-order array combinator (SOAC, see Section 2.1.3 below).

- *Second-order array language*: this term makes the most sense applied to domain-specific languages for arrays specifically; it indicates that while there are some built-in operations that take user-written functions, lambdas may syntactically be written only as an argument to such built-ins. For example, the following code would be valid for building a new array with each element one larger than the corresponding element in the array a:

$$\text{map } (\lambda x.\ x + 1)\ a \tag{2.1}$$

  but the following is disallowed:

$$\textbf{let } f = \lambda x.\ x + 1 \textbf{ in } \text{map } f\ a \tag{2.2}$$

  This definition of a second-order array language may strike the reader as somewhat awkward; surely Eq. (2.2) can be automatically inlined to Eq. (2.1) by a compiler — why prohibit it? The motivation is that we wish to disallow Eq. (2.3):

$$\begin{array}{l}\textbf{let } f = \textbf{if } \ldots \textbf{ then } \lambda x.\ x + 1 \textbf{ else } \lambda x.\ y \cdot x \\ \textbf{in } \text{map } f\ a\end{array} \tag{2.3}$$

because returning lambda functions from conditionals in general makes it unknown at compile time which lambda function (and thus which closure object!) ends up in which array combinator. Knowing this data flow precisely turns out to be important for good code generation, as well as for efficient compile-time reverse AD.[1]

In practice, some second-order array languages (e.g. Accelerate) have internal compiler representations that strictly conform to the definition, but allow forms like Eq. (2.2) in the source language for convenience.

- *Higher-order language*: lambda abstraction is fully supported; functions can be created, applied, passed around, and put into and retrieved from data structures at will. Such languages are very expressive — perhaps too expressive, as programmers can essentially define their own control flow structures, obscuring the structure of the program from the compiler and thereby potentially depriving it of the information it needs to efficiently differentiate and compile the program to parallel hardware. We support higher-order languages in Chapter 3 on dual-numbers reverse AD and in Chapters 5 and 6 on CHAD, but revert to less expressive languages in the more performance-oriented Chapters 4 and 7.

We also have an accompanying notion of data without function values inside:

- *Zeroth-order type*: a data type that does not contain function types, neither directly nor in any user-defined data types mentioned in the type, in case the language supports those. Values of zeroth-order type are referred to as *zeroth-order data*. Confusingly, such values are also known as *first-order values* in literature, but for consistency with first-order operations (that take zeroth-order arguments) and second-order array combinators (that can take first-order arguments) from above, we use "zeroth-order" in this thesis.

  Most array languages only support zeroth-order arrays, regardless of the order of the rest of the language.

Note that this terminology of "order" is distinct from "rank" in type theory, which refers to the placement of $\forall$-quantifiers in types. A *higher-rank type* is one with a $\forall$-quantifier that is not at the start of the type, such as $(\forall s.\ \mathsf{ST}\ s\ a) \to a$ (the type of `runST` in Haskell); such types are out of scope in this thesis.

[1] Of course, Eq. (2.3) itself can be turned into the form of Eq. (2.2) by commuting **if** and λ, but in general (if the branches do more than directly return a closure) this introduces work duplication.

$$
\begin{array}{ll}
\rho ::= \mathbb{R} \mid \mathbb{Z} & \text{(element types)} \\
\tau ::= \rho \mid \text{Vec } \rho & \text{(types)} \\
t ::= x \mid \textbf{let } x = t^{\tau_1} \textbf{ in } t^{\tau_2} & \text{(variables, bindings)} \\
\quad \mid 0 \mid 1 \mid \ldots \mid [t^\rho, \ldots, t^\rho] & \text{(literals)} \\
\quad \mid \text{length } t^{\text{Vec } \rho} & \text{(array shape query)} \\
\quad \mid t^\tau + t^\tau \mid t^\tau - t^\tau \mid t^\tau \cdot t^\tau \mid \sin t^\tau \mid \ldots & \text{(arithmetic operations)} \\
\quad \mid \text{sum } t^{\text{Vec } \rho} \mid \text{product } t^{\text{Vec } \rho} & \text{(reductions)} \\
\quad \mid \text{prefixSums } t^{\text{Vec } \rho} & \text{(scans)} \\
\quad \mid t^{\text{Vec } \rho} \,!\, t^{\mathbb{Z}} \mid \text{gather } t^{\text{Vec } \rho}\ t^{\text{Vec } \mathbb{Z}} & \text{(indexing, backward perm.)} \\
\qquad \text{— } \textit{e.g.}\ [2,4,7] \,!\, 1 = 4\textit{;}\ \text{gather } [2,4,7]\ [1,1,2,0] = [4,4,7,2] & \\
\quad \mid \text{scatter } t^{\mathbb{Z}}\ t^{\text{Vec } \rho}\ t^{\text{Vec } \mathbb{Z}} & \text{(forward perm.)} \\
\qquad \text{— } \textit{e.g.}\ \text{scatter } 5\ [2,4,7,8]\ [1,0,3,0] = [12,2,0,7,0]\ \textit{(note: } 4+8=12\textit{)} & \\
\quad \mid \textbf{if } t^{\mathbb{Z}} \textbf{ then } t^\tau \textbf{ else } t^\tau & \text{(conditionals (if-not-zero))}
\end{array}
$$

Figure 2.1: A simplistic first-order array language with 1-dimensional arrays ('Vec'). Arithmetic operations work elementwise on arrays. Subterm types are indicated with superscripts: τ, ρ stand for arbitrary types of their respective syntactic categories. Equal names indicate that types must be equal.

2.1.2 First-order array languages

In a first-order array language, all built-in operations are first-order: they do not take functions as arguments. This means that operations like map : $(a \to b) \to \text{Array } a \to \text{Array } b$, to lift a function on elements to work elementwise over arrays, do not exist in the language.

A hypothetical simplistic first-order array language is presented in Fig. 2.1. It misses specialised primitives for e.g. array transpose, and with its lack of support for higher-dimensional arrays, matrix computations are awkward. Furthermore, it lacks a sequential looping construct. Nevertheless, simple computations can be written in it, and the operations in this mini-language do form the basis of typical first-order array languages.

Purely first-order array languages are rare; ArrayFire[2] is one language of this type. However, a number of other functional array languages, even if they technically have some second-order functions, are commonly seen as first-order because in the majority of use cases, those second-order functions are never used; such languages include NumPy[3], TensorFlow [Abadi et al. 2016] and PyTorch [Paszke et al. 2017]. Note that while some of these, especially NumPy and PyTorch, are

[2] `https://arrayfire.com`; `https://arrayfire.org/docs`

[3] `https://numpy.org` (e.g. `numpy.apply_along_axis` takes a function argument)

$$
\begin{array}{lll}
\rho, \rho_1, \rho_2 \coloneqq \mathbb{R} \mid \mathbb{Z} \mid (\rho_1, \rho_2) & & \text{(element types)} \\
\tau, \tau_1, \tau_2 \;\; \coloneqq \mathbb{R} \mid \mathbb{Z} \mid (\tau_1, \tau_2) \mid \text{Vec } \rho & & \text{(types)} \\
t \coloneqq x \mid \textbf{let } x = t^{\tau} \textbf{ in } t^{\tau} & & \text{(variables, bindings)} \\
\quad \mid (t^{\tau_1}, t^{\tau_2}) \mid \text{fst } t^{(\tau_1,\tau_2)} \mid \text{snd } t^{(\tau_1,\tau_2)} & & \text{(product types)} \\
\quad \mid 0 \mid 1 \mid \ldots \mid [t^{\rho}, \ldots, r^{\rho}] & & \text{(literals)} \\
\quad \mid \text{length } t^{\text{Vec } \rho} & & \text{(array shape query)} \\
\quad \mid t^{\rho} + t^{\rho} \mid t^{\rho} - t^{\rho} \mid t^{\rho} \cdot t^{\rho} \mid \sin t^{\rho} \mid \ldots & & \text{(arithmetic operations)} \\
\quad \mid t^{\text{Vec } \rho} \,!\, t^{\mathbb{Z}} & & \text{(indexing)} \\
\quad \mid \text{build } t^{\mathbb{Z}} \; (\lambda x^{\mathbb{Z}}.\; t^{\rho}) & & \text{(elementwise construction)} \\
\qquad \text{— } e.g. \text{ build } 4 \; (\lambda i.\; i \cdot i) = [0, 1, 4, 9] & & \\
\quad \mid \text{foldl } (\lambda x^{\rho} \, y^{\rho}.\; t^{\rho}) \; t^{\rho} \; t^{\text{Vec } \rho} & & \text{(associative reduction)} \\
\qquad \text{— } e.g. \text{ foldl } (\lambda x \, y.\; x + y) \; 0 \; [1, 2, 3, 4] = 0 + 1 + 2 + 3 + 4 = 10 & & \\
\quad \mid \text{scanl } (\lambda x^{\rho} \, y^{\rho}.\; t^{\rho}) \; t^{\rho} \; t^{\text{Vec } \rho} & & \text{(associative scan)} \\
\qquad \text{— } e.g. \text{ scanl } (\lambda x \, y.\; x + y) \; 0 \; [1, 2, 3, 4] = [0, 1, 3, 6, 10] & & \\
\quad \mid \text{scatter } (\lambda x^{\rho} \, y^{\rho}.\; t^{\rho}) \; t^{\text{Vec } \rho} \; (\lambda x^{\mathbb{Z}}.\; t^{\mathbb{Z}}) \; t^{\text{Vec } \rho} & & \text{(forward permutation)} \\
\qquad \text{— } e.g. \text{ scatter } (\lambda x \, y.\; x + y) \; [0, 0, 0, 0] \; (\lambda i.\; i \bmod 3) \; [4, 7, 6, 1] = [5, 7, 6, 0] & & \\
\qquad\quad \textit{(note: } 4 + 1 = 5\textit{)} & & \\
\quad \mid \textbf{if } t^{\mathbb{Z}} \textbf{ then } t^{\tau} \textbf{ else } t^{\tau} & & \text{(conditionals (if-not-zero))}
\end{array}
$$

Figure 2.2: A simple second-order array language with 1-dimensional, non-nested arrays ('Vec'). A derived operation is map $f \; a$ = build (length a) $(\lambda i.\; f \; (a \,!\, i))$. Like in Fig. 2.1, superscript annotations are types. The intent of the notation is that the ρ types are a subset of the τ types; e.g. one can use 'fst' on an element projected from an array.

typically described as *libraries* rather than languages, one can easily see them as *embedded DSLs* that inherit the binding and control flow structures of the host language — Python, in this case.

2.1.3 Second-order array combinators (SOACs)

While first-order array languages are perfectly serviceable and relatively easy to compile (or differentiate), we can lean more into the functional approach to language design by providing fewer, more powerful primitive combinators.[4]

An example language, inspired by Accelerate, is given in Fig. 2.2. Note that in addition to replacing the first-order array operations of Fig. 2.1 with *second-order array combinators* (SOACs) like 'build', 'foldl', etc. (and thereby making some

[4]These functions (SOACs) are often called *combinators*, not operations, because they compose in more ways than just saturated function application. They are just language built-ins, however.

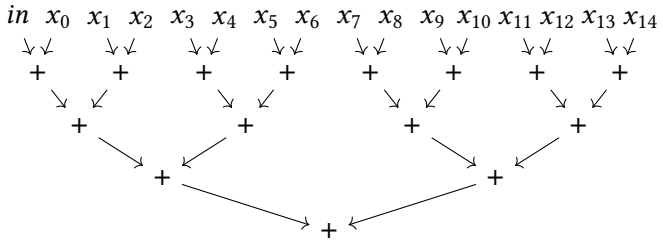


Figure 2.3: A naive tree reduction to compute the sum of an array. The initial element (the second argument of 'foldl' in Fig. 2.2) is '*in*'.

operations like 'prefixSums' and 'gather' redundant), we have also added *product types*[5] to the type system. While the presence of product types in the type system is in principle orthogonal to the style of array primitives, programs written using SOACs do more in a single array operation, and hence sometimes end up having more than one output from a SOAC per array element. Supporting arrays of products makes this possible.

For the 'foldl' and 'scanl' SOACs, the word "associative" in Fig. 2.2 refers to the fact that the combination function (their first argument) must be an associative function, i.e. $f\ x\ (f\ y\ z) = f\ (f\ x\ y)\ z$. This allows the values in the input array to be reduced in any order, enabling a parallel implementation with *span* (the length of the longest dependency chain; see Section 2.1.5 below) equal to $\Theta(\log(n))$, where n is the length of the input array. For a reduction ('foldl'), one can apply a tree reduction like in Fig. 2.3, and for a scan (e.g. 'scanl'), a naive option is shown schematically in Fig. 2.4. Practical algorithms, especially on GPUs, are more sophisticated (e.g. the decoupled look-back scan of Merrill and Garland [2016]).

Some examples of functional array languages that are designed around SOACs are Futhark[6] [Henriksen 2017], Accelerate [Chakravarty et al. 2011], SaC[7] [Scholz 2003] and Lift [Steuwer et al. 2017]. Generally speaking, the purpose of these languages is to produce efficient machine code from a very structured, high-level language, freeing the programmer from low-level optimisation concerns like whether two loops ought to be fused together; what the latest research is on parallel scan algorithms; and other implementation concerns. The purpose is usually *not* to compete with expert-written and expert-optimised code in C, Fortran, CUDA or similar: the aim is to be faster than non-expert code in such languages, while simultaneously offering a much faster feedback cycle

[5] Also known as: tuples (when n-ary); pairs (when binary); structs, records (when labeled). The ones in Fig. 2.2 are binary, i.e. with two fields.

[6] `https://futhark-lang.org`

[7] `https://www.sac-home.org`

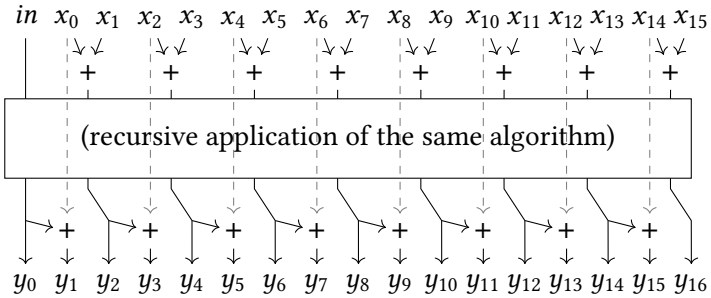


Figure 2.4: A (very) naive algorithm to compute prefix sums of an array in parallel. The initial element (the second argument of 'scanl' in Fig. 2.2) is again '*in*'.

during development. To benefit from already-written code in languages with more manual control, functional array languages tend to offer a *foreign-function interface* (FFI) that allows calling out to code in a different language from the context of a larger, functional program.

Special-case combinators. Despite the design principle of having fewer, more general combinators, second-order array languages do often also have some built-in array operations that are special cases of those general combinators; for example, aside from a general reduction combinator like 'foldl' from Fig. 2.2, Futhark also has a fold that requires the combination function to be *commutative*, as this allows a more efficient parallel reduction algorithm.[8] While such patterns (like commutativity of a user-written function) could also, to a certain extent, be automatically detected in uses of the more general combinators, this is not straightforward outside of trivial cases and compiler developer time is often better spent elsewhere.[9]

Unlike first-order languages, however, second-order languages typically have much fewer of these special-case combinators, because usually (though not quite always), general code generation approaches can be devised that subsume the optimisations that otherwise only the special-case combinator would enjoy. This holds for operations like 'sum', 'map', 'zipWith', 'gather', 'transpose', and more. [de Wolff et al. 2025] However, because such combinators have more explicit data and control flow than the corresponding more general combinators, having (or at least recognising) these combinators in user code can be very advantageous for AD; for example, 'transpose' is just a special case of 'build', but the reverse derivative of 'build' is rather complex and does not easily reduce back to the

[8]The `reduce_comm` combinator: `https://futhark-lang.org/docs/prelude/doc/prelude/soacs.html#term:reduce_comm`

[9]Such special-case detection also quickly runs afoul of Rice's theorem.

reverse derivative of 'transpose', which is simply 'transpose'.

Language extensions. As with the first-order language in Fig. 2.1, the second-order array language in Fig. 2.2 — while quite capable already — lacks some essential features like sequential loops and multi-dimensional arrays. Practical languages typically do have such functionality, either natively or inherited from the host language (when embedded).

2.1.4 Optimisations on second-order array languages

Some of the typical optimisations in compilers for second-order functional array languages are important enough that they have an influence on how to write fast code in those languages — and thus also on how to do efficient AD on them. We discuss two here: struct-of-arrays representation and loop fusion.

Struct-of-arrays. *Struct-of-arrays* (SoA) representation, or *struct-of-arrays form*, refers to a particular layout of large amounts of uniformly structured data, and contrasts with *array-of-structs* (AoS) representation.[10] If one has a list of, say, points in 3D space, a natural representation for that list would be an array of tuples: Vec $(\mathbb{R}, \mathbb{R}, \mathbb{R})$ in the notation of Figs. 2.1 and 2.2. This representation (the array-of-structs form) groups the coordinates of a single point close to each other in memory. An alternative representation is $(\text{Vec}\ \mathbb{R}, \text{Vec}\ \mathbb{R}, \text{Vec}\ \mathbb{R})$, which instead groups all x-coordinates together, all y-coordinates together, etc.; this is the struct-of-arrays form.

If computations act uniformly on all points in parallel, SoA typically allows much superior performance: one can use SIMD instructions[11] to operate on multiple points in parallel for the price of one, replacing e.g. an x scalar in a computation by a vector of, say, eight x values. Such a *vectorised* computation essentially works on data in SoA form, so if the data is stored in memory in AoS form instead, it needs to be converted back and forth to do the vectorised computation. The rearrangement of values in, and between, hardware vector registers that is necessary for this conversion can be expensive enough to negate much of the performance advantage of vectorisation.

[10] The names, in particular the use of "struct", come from imperative programming.

[11] Single-instruction multiple-data, also known as *vector instructions*. These are designed to perform the same operation in parallel on multiple homogeneous values; communication between vector components ("lanes") is generally awkward and somewhat slow. To get good speedups from SIMD with AoS representation, the computation on a *single* point would need to act uniformly *elementwise* on all coordinates of a point (and the point should really have a multiple of 4 coordinates). Most interesting computations on points violate this, e.g. (squared) vector norm $(x^2 + y^2 + z^2)$.

While C compilers are bound by the C standard to lay out structure fields almost exactly as the programmer specified, higher-level languages have no such restrictions; as such, functional array languages that support arrays of product types almost invariably represent those transparently in SoA form at runtime. Thus, even if one writes 'Vec $(\mathbb{R}, \mathbb{R}, \mathbb{R})$' in one's code, the compiled program will in fact work on the transposed representation $(\text{Vec}\ \mathbb{R}, \text{Vec}\ \mathbb{R}, \text{Vec}\ \mathbb{R})$ instead.

Compiling to a SoA representation also means that zipping primitives, if the language provides them:

$$\begin{aligned}\text{zip} &: \text{Vec}\ a \to \text{Vec}\ b \to \text{Vec}\ (a, b)\\ \text{unzip} &: \text{Vec}\ (a, b) \to (\text{Vec}\ a, \text{Vec}\ b)\end{aligned}$$

are typically zero-cost at runtime. A caveat is that in case of support for (rectilinear) multi-dimensional arrays, the 'zip' operation may need to allocate a rearranged array at runtime if its operands do not have equal shapes.

Loop fusion. The most characteristic optimisation for array languages is *fusion*, also known as *deforestation* in the functional programming literature. The idea is that the following function:

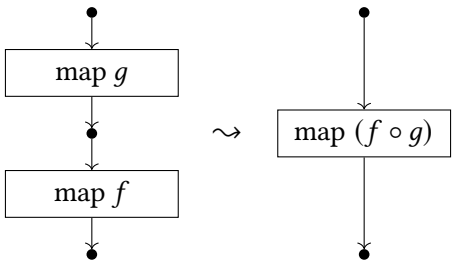
$$\lambda a.\ \text{map}\ f\ (\text{map}\ g\ a) \tag{2.4}$$

is semantically equivalent to:

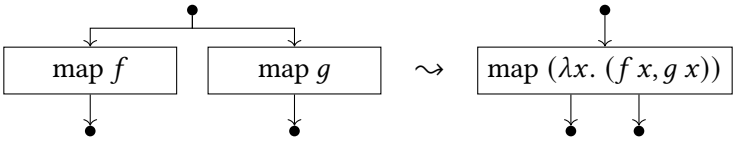
$$\lambda a.\ \text{map}\ (f \circ g)\ a \tag{2.5}$$

and that furthermore, the latter is much more efficient than the former: aside from saving the overhead of one parallel loop (iteration administration and scheduling), it eliminates the intermediate array and with it one memory write and one memory read operation per element. If f and g are relatively cheap functions, the bottleneck of Eq. (2.4) was likely memory bandwidth rather than computation capacity (such computations are called *memory-bound*; the dual, and typically more desirable, situation is called *compute-bound*), meaning that eliminating memory traffic can significantly speed up the program.

Moving from Eq. (2.4) to Eq. (2.5) is called *vertical fusion*, from its appearance in a top-to-bottom data flow graph:



where we draw manifest arrays in memory as •, and computation as a box. Another common variant is *horizontal fusion*:



Here, the right-hand form is equivalent to the left-hand form if one assumes that arrays are in struct-of-arrays form; it saves the overhead of one loop, plus one memory read (of the source array) per element. It can also enable further vertical fusion.

The examples here fuse two 'map' combinators, but various other combinations are also fusible; for example, if the language has a dedicated second-order 'gather' combinator with the following syntax and (sketched) semantics:

$$t ::= \ldots \mid \text{gather}\ t^{\mathbb{Z}}\ (\lambda x^{\mathbb{Z}}.\ t^{\mathbb{Z}})\ t^{\text{Vec}\ \rho} \mid \ldots$$
$$\text{— } \textit{e.g.}\ \text{gather}\ 3\ (\lambda i.\ i+1)\ [3,5,4,1,2] = [5,4,1]$$

then the following term:

$$\text{gather}\ k\ (\lambda i.\ i \bmod n)\ (\text{gather}\ n\ (\lambda i.\ i+1)\ a)$$

fuses to:

$$\text{gather}\ k\ (\lambda i.\ (i \bmod n)+1)\ a$$

Finally, while combinations like foldl $(\lambda x\, y.\ x+y)\ 0\ (\text{map}\ f\ a)$ cannot be simplified to a single array combinator from Fig. 2.2, the (parallel) loops can certainly be fused at code generation time; array languages accomplish this by having a more expressive internal language that can express such more advanced clusters.

For examples and more reading on fusion in functional array languages, see e.g. [McDonell et al. 2013; van Balen et al. 2024] on Accelerate and [Henriksen et al. 2017] on Futhark.

2.1.5 Parallelism and complexity

When reasoning at a high level about parallel algorithms (in the context of array languages or otherwise), one is immediately interested in analysing their computational complexity. Standard big-O notation for computational (and memory) complexity of sequential algorithms can be extended to parallel algorithms by introducing the concepts of *work* and *span*.[12] The work of an algorithm is the time taken to execute it on a single processor; the span is the time taken on an

[12] See e.g. [Blelloch 1996] for another description; the terms there are 'work' and 'depth'.

infinite number of processors. (Inconveniences like communication time between these processors are disregarded.)

For example, any sequential algorithm has work and span both equal to its conventional time complexity. The program 'map $(\lambda x.\ x + 1)\ a$' has work $W = \Theta(n)$ and span $S = \Theta(1)$, where n is the length of the array a. The reduction and scanning algorithms shown in Figs. 2.3 and 2.4 both have work $\Theta(n)$ and span $\Theta(\log(n))$. Because of this logarithmic factor in the span of a reduction, and in particular of the reverse derivative of replication (with span $\Theta(1)$), which is a sum (with span $\Theta(\log(n))$), reverse AD on parallel array programs cannot be fully complexity-preserving.

2.2 Automatic Differentiation

This section has two explicit goals:

1. Give an introduction to AD for an audience familiar with functional programming, that gives intuition about what exactly is being computed and why the algorithms look the way they do;

2. Give an overview of the literature on AD that is most relevant to this thesis.

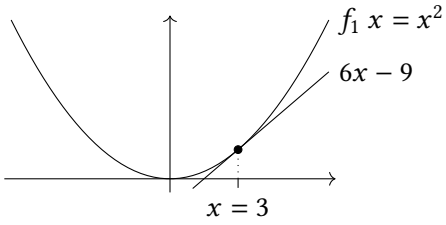
In service of goal 1, we start by giving a rather opinionated exposition of multivariate derivatives; afterwards, we look at a principled, yet simple and visual, intuition for how AD algorithms implement these definitions. Subsequently, from Section 2.2.8, we connect this intuition to (efficient) AD implementation techniques and, for goal 2, the associated literature.

Readers already familiar with multivariate derivatives and Jacobian matrices may skip Sections 2.2.1 and 2.2.2; just note the notation $Df\ \vec{x}\ \vec{u} = Jf\ \vec{x} \cdot \vec{u}$ (the *forward derivative* / total derivative / pushforward) and $(Df)^\top\ \vec{x}\ \vec{u} = (Jf\ \vec{x})^\top \cdot \vec{u}$ (its transpose, the *reverse derivative* / pullback). Readers already familiar with dual-numbers forward AD may additionally skip Section 2.2.3.

2.2.1 Derivatives

The derivative of a function $f : \mathbb{R} \to \mathbb{R}$ describes how its output responds to small changes, or *perturbations*, of its input. In other words, it describes a function's instantaneous rate of change. For example, take the function $f_1 : \mathbb{R} \to \mathbb{R}$ given by $f_1\ x = x^2$; the tangent line to this function's graph[13] at the input $x = 3$ has slope 6:

[13] For space-efficiency, the plot actually shows $\frac{1}{7}x^2$.



This means that the *partial derivative* of f_1 with respect to x, at $x = 3$, equals 6:[14]

$$\frac{\partial f_1}{\partial x}(3) = 6$$

Note that this value indeed expresses how f_1 responds to perturbations of its input: if we evaluate f_1 at $x = 3 + \varepsilon$ for a very small ε, the result is $9 + 6\varepsilon + O(\varepsilon^2) \approx 9 + 6\varepsilon$. That is to say: specifically at the point $x = 3$, a small change to f_1's input results in a change of f_1's output that is 6 times as large.[15] Said differently, at $x = 3$ (hence "instantaneous"), the output of f_1 changes 6× as fast as its input. In the perturbation computation using ε, we ignore higher powers of ε because we aim to capture the *linear* effect that a change in f_1's input has on its output; higher-order (superlinear) effects are captured by second, third etc. derivatives of f_1.

Because we have a formula for f_1, we can even give its derivative symbolically: $\frac{\partial f_1}{\partial x}(x_0) = 2x_0$. With this formula in hand, we can also generalise the above perturbation computation: if we evaluate f_1 at $x = x_0 + \varepsilon$ for very small ε, the result is $x_0^2 + 2x_0\varepsilon + O(\varepsilon^2) \approx x_0^2 + 2x_0\varepsilon = f_1\ x_0 + 2x_0\varepsilon$.

One can also obtain a partial derivative using its mathematical definition, a limit of a difference quotient:

$$\frac{\partial f_1}{\partial x}(x_0) = \lim_{h \to 0} \frac{f_1\ (x_0 + h) - f_1\ x_0}{h} = \lim_{h \to 0} (2x_0 + h) = 2x_0$$

However, the formulation using perturbations turns out to be more useful for obtaining a compositional algorithm for constructing the derivative of a *program*, i.e. a function written as code in a programming language. Specifically, in this thesis, we will not consider the derivative of a function to be its instantaneous rate of change as a numeric value; instead, we will consider it to be another *function* that takes a perturbation of the input to the resulting change in the output. In

[14] Traditionally, one uses total derivative notation ($\frac{\mathrm{d}f}{\mathrm{d}x}$) if x is f's only argument, and partials ($\frac{\partial f}{\partial x}$) otherwise. Because (1) a 1-argument function is nothing more than an n-argument function for which $n = 1$, (2) the distinction is not helpful in this exposition, and (3) the notion of "only argument" becomes somewhat muddy in Section 2.2.4 and onwards, we use "partial"/∂ consistently.

[15] In this section we will use the terminology of "small changes" more often. To put this on a rigorous mathematical footing, one can replace these by *infinitesimals* as in Robinson's nonstandard analysis, and insert standard-part conversions in the appropriate places.

the case of f_1, we write this *(forward) derivative function*, also known as the *total derivative*, as:

$$\begin{aligned} &Df_1 : \mathbb{R} \to \underline{\mathbb{R}} \to \underline{\mathbb{R}} \\ &Df_1\ x\ u = 2xu \end{aligned}$$

using u, instead of ε, as the variable for the perturbation of the input. We write $\underline{\mathbb{R}}$ for the type of the "rate of change" of a real value; mathematically this is the *tangent space* to $\mathbb{R}$, but since it is isomorphic to $\mathbb{R}$, one can simply use $\underline{\mathbb{R}} \coloneqq \mathbb{R}$.

Note that for all x, $Df_1\ x$ is a *linear function* between vector spaces. In the subsequent generalisations, this will remain true.

Aside: Fréchet derivative. For differentiable functions $f : \mathbb{R}^m \to \mathbb{R}^n$, our forward derivative function Df coincides with f's Fréchet derivative in mathematics. A function $f : \mathbb{R}^m \to \mathbb{R}^n$ is said to be *Fréchet differentiable*, or simply *differentiable*, if there exists a function $Df : \mathbb{R}^m \to \underline{\mathbb{R}}^m \to \underline{\mathbb{R}}^n$ (its *Fréchet derivative*) such that the second arrow is linear (i.e. $Df\ \vec{x} : \underline{\mathbb{R}}^m \to \underline{\mathbb{R}}^n$ is a linear function for all $\vec{x}$) and the following equation is satisfied:

$$\lim_{\vec{h} \to \vec{0}} \frac{\|f\ (\vec{x} + \vec{h}) - (f\ \vec{x} + Df\ \vec{x}\ \vec{h})\|_{\mathbb{R}^n}}{\|\vec{h}\|_{\mathbb{R}^m}} = 0$$

Essentially, this equation says that $Df\ \vec{x}$ is the best local linear approximation of f around $\vec{x}$. In the remainder of this chapter, unless otherwise specified, we will assume that all input functions are differentiable at least on the domain on which we evaluate them, so that we can simply speak about perturbations without needing to check the limit condition.[16]

More inputs. Functions of type $\mathbb{R}^m \to \mathbb{R}$ are harder to plot than simple $\mathbb{R} \to \mathbb{R}$ functions, especially if $m > 2$, but the algebra does not mind. Consider a new example function $f_2 : \mathbb{R}^2 \to \mathbb{R}$ given by $f_2\ (x, y) = x^2 + 3xy$. Where f_1 from above had one partial derivative, f_2 has two:

$$\frac{\partial f_2}{\partial x}(x, y) = 2x + 3y \qquad \frac{\partial f_2}{\partial y}(x, y) = 3x$$

These are commonly bundled in the *gradient* of f_2:

$$\begin{aligned} &\nabla f_2 : \mathbb{R}^2 \to \underline{\mathbb{R}}^2 \\ &\nabla f_2\ (x, y) = \left(\frac{\partial f_2}{\partial x}(x, y), \frac{\partial f_2}{\partial y}(x, y)\right) \end{aligned}$$

[16] Technically, AD implementations assume even more in the presence of conditionals; see e.g. [Hückelheim et al. 2023, §3.3] for more discussion.

The gradient of a function f at $\vec{x}$ is a vector that points in the direction in which f changes fastest, starting from $\vec{x}$; the norm (length) of the gradient indicates *how* fast f changes in that direction.

Like before, we can also construct the *forward derivative* of f_2. What should its type be? The forward derivative at a point should show, given a perturbation to (each component of) the input, how the output of f_2 changes in response.

$$\begin{aligned} &Df_2 : \mathbb{R}^2 \to \underline{\mathbb{R}^2} \to \underline{\mathbb{R}} \\ &Df_2\ (x,y)\ (u,v) = ? \end{aligned}$$

How *does* the output of f_2 change? Well, if x changes by (a small quantity) u, then $f_2\ (x,y)$ changes by $\frac{\partial f_2}{\partial x}(x,y) \cdot u$; this is the same as the single-dimensional case discussed before.[17] Analogously, if y changes by v, then $f_2\ (x,y)$ changes by $\frac{\partial f_2}{\partial y}(x,y) \cdot v$. Because f_2 is *differentiable*, the output of f_2 is simply subject to both changes simultaneously — and hence changes by their sum:

$$\begin{aligned} &Df_2 : \mathbb{R}^2 \to \underline{\mathbb{R}^2} \to \underline{\mathbb{R}} \\ &Df_2\ (x,y)\ (u,v) = \frac{\partial f_2}{\partial x}(x,y) \cdot u + \frac{\partial f_2}{\partial y}(x,y) \cdot v \\ &\qquad\qquad\quad = \nabla f_2\ (x,y) \cdot (u,v) \end{aligned} \tag{2.6}$$

writing $(\cdot) : \mathbb{R}^m \to \mathbb{R}^m \to \mathbb{R}$ for the dot product on the last line. (Note that, as expected, $Df_2\ \vec{x}$ is a linear function for all $\vec{x}$.)

More outputs. Now suppose we have the following function:

$$\begin{aligned} &f_3 : \mathbb{R}^3 \to \mathbb{R}^2 \\ &f_3\ (x,y,z) = (\underbrace{x^2z + 3xy}_{(f_3)_1}, \underbrace{xz^2 + 4y^2}_{(f_3)_2}) \end{aligned}$$

This function has *six* partial derivatives, which we can suggestively arrange in a matrix:

$$Jf_3\ (x,y,z) = \begin{pmatrix} \frac{\partial (f_3)_1}{\partial x}(x,y,z) & \frac{\partial (f_3)_1}{\partial y}(x,y,z) & \frac{\partial (f_3)_1}{\partial z}(x,y,z) \\ \frac{\partial (f_3)_2}{\partial x}(x,y,z) & \frac{\partial (f_3)_2}{\partial y}(x,y,z) & \frac{\partial (f_3)_2}{\partial z}(x,y,z) \end{pmatrix} = \begin{pmatrix} 2xz + 3y & 3x & x^2 \\ z^2 & 8y & 2x \end{pmatrix}$$

This matrix is called the *Jacobian matrix* of f_3 at the point (x,y,z); the type of Jf_3 is $\mathbb{R}^3 \to \underline{\mathbb{R}^{2\times 3}}$. More generally, the rows of the Jacobian matrix of a function are the gradients of its components (in this case, of $(f_3)_1$ and $(f_3)_2$).

Let us again write down the forward derivative. In terms of perturbations, each component of f_3 can be seen as an separate function that is independently

[17] Indeed, one can write $D(x \mapsto f_2\ (x,y))\ x\ u = \frac{\partial f_2}{\partial x}(x,y) \cdot u$.

influenced by changing the inputs to f_3. Hence, we essentially get Eq. (2.6) elementwise over the output of f_3:

$$
\begin{aligned}
&Df_3 : \mathbb{R}^3 \to \underline{\mathbb{R}}^3 \to \underline{\mathbb{R}}^2 \\
&Df_3\ (x,y,z)\ (u,v,w) = \big(\nabla(f_3)_1\ (x,y,z) \cdot (u,v,w), \nabla(f_3)_2\ (x,y,z) \cdot (u,v,w)\big) \\
&\quad = \begin{pmatrix} \text{——}\ \nabla(f_3)_1\ (x,y,z)\ \text{——} \\ \text{——}\ \nabla(f_3)_2\ (x,y,z)\ \text{——} \end{pmatrix} \cdot \begin{pmatrix} u \\ v \\ w \end{pmatrix} = Jf_3\ (x,y,z) \cdot \begin{pmatrix} u \\ v \\ w \end{pmatrix}
\end{aligned}
$$

The pattern that we observe here works in general (if f is differentiable at $\vec{x}$):

$$Df\ \vec{x}\ \vec{u} = Jf\ \vec{x} \cdot \vec{u} \tag{2.7}$$

This equation (2.7) also makes sense from the perspective of types: a matrix in $\underline{\mathbb{R}}^{2\times3}$ indeed corresponds to a linear function $\underline{\mathbb{R}}^3 \to \underline{\mathbb{R}}^2$. In general, we say that the forward derivative of a function is a *Jacobian–vector product*. Indeed, a common name for the general forward derivative in AD implementations is JVP.[18]

In particular, it is instructive to verify that Eq. (2.7) also applied to f_2 and f_1 that we analysed earlier, when seen as functions $f_2 : \mathbb{R}^2 \to \mathbb{R}^1$ and $f_1 : \mathbb{R}^1 \to \mathbb{R}^1$.

Another perspective on the forward derivative is that it computes a linear combination of the columns of the Jacobian of a function; the coefficients of the linear combination are in the tangent argument ($\vec{u}$). If the Jacobian only has one column (i.e. the domain of the function has dimensionality 1), then passing $\vec{u} = \big(1\big)$ to the forward derivative computes the entire Jacobian.

Spoiler: forward AD. The forward mode of AD computes this Jacobian–vector product (i.e. Df) with only a constant-factor overhead over the original function in time and in space. That is to say: given a program (term) t that computes a function f, forward AD turns t into a program t' that computes the function Df, and if evaluating t on an input $\vec{x}$ takes time T and memory M, then evaluating t' on $\vec{x}$ and $\vec{u}$ takes time in $O(T)$ and memory in $O(M)$. There is a brief discussion of a simple (yet efficient) algorithm for forward AD in Section 2.2.3.

2.2.2 Reverse derivatives

Functions that we want to compute derivatives of, often have type $\mathbb{R}^m \to \mathbb{R}$: for example, the training objective of a neural network has many parameters and just a single scalar output (the loss function). For such a function, computing the

[18] For example in JAX: `https://docs.jax.dev/en/latest/_autosummary/jax.jvp.html` (accessed 2025-04-02, preserved on `https://web.archive.org`)

full Jacobian using Df would require m evaluations:

$$Df\ \vec{x}\ e_i = \left(\frac{\partial f}{\partial x_1}(\vec{x}) \quad \cdots \quad \frac{\partial f}{\partial x_m}(\vec{x})\right) \cdot e_i = \frac{\partial f}{\partial x_i}(\vec{x})$$
$$\text{thus: } \left(Df\ \vec{x}\ e_1 \quad \cdots \quad Df\ \vec{x}\ e_m\right) = \left(\frac{\partial f}{\partial x_1}(\vec{x}) \quad \cdots \quad \frac{\partial f}{\partial x_m}(\vec{x})\right) = Jf\ \vec{x}$$

where $e_i = (0, \ldots, 0, 1, 0, \ldots, 0)^\top \in \mathbb{R}^m$ with the 1 at the i'th position. For this reason, training a neural network with any realistic number of parameters using forward AD would be extremely slow.

However, matrices can be transposed. By analogy with the forward derivative, for an $f : \mathbb{R}^m \to \mathbb{R}^n$ we can thus also define (for now, just in the mathematical world) the *reverse derivative*:

Forward derivative	**Reverse derivative**
$Df : \mathbb{R}^m \to \underline{\mathbb{R}^m} \to \underline{\mathbb{R}^n}$	$(Df)^\top : \mathbb{R}^m \to \underline{\mathbb{R}^n} \to \underline{\mathbb{R}^m}$
$Df\ \vec{x}\ \vec{u} = Jf\ \vec{x} \cdot \vec{u}$	$(Df)^\top\ \vec{x}\ \vec{u} = (Jf\ \vec{x})^\top \cdot \vec{u}$

The reverse derivative is less directly connected to the propagation of "small changes" than the forward derivative is, but we can make sense of what it computes through the notion of the gradient; as we have already seen, the gradient (∇) of a function $g : \mathbb{R}^m \to \mathbb{R}$ describes in what direction one should move from some input $\vec{x}$ to make g's output change fastest. Now, because the rows of the Jacobian matrix of f are precisely the gradients of the components of f:

$$Jf\ \vec{x} = \begin{pmatrix} \frac{\partial f_1}{\partial x_1}(\vec{x}) & \cdots & \frac{\partial f_1}{\partial x_m}(\vec{x}) \\ \vdots & \ddots & \vdots \\ \frac{\partial f_n}{\partial x_1}(\vec{x}) & \cdots & \frac{\partial f_n}{\partial x_m}(\vec{x}) \end{pmatrix} = \begin{pmatrix} \text{——}\ \nabla f_1\ \vec{x}\ \text{——} \\ \vdots \\ \text{——}\ \nabla f_n\ \vec{x}\ \text{——} \end{pmatrix}$$

the reverse derivative of f computes a linear combination of the gradients of the components of f:

$$(Df)^\top\ \vec{x}\ \vec{u} = \begin{pmatrix} | & & | \\ (\nabla f_1\ \vec{x})^\top & \cdots & (\nabla f_n\ \vec{x})^\top \\ | & & | \end{pmatrix} \cdot \vec{u} = u_1 \cdot (\nabla f_1\ \vec{x}) + \cdots + u_n \cdot (\nabla f_n\ \vec{x}) \tag{2.8}$$

Thus, where the forward derivative computed a linear combination of the *columns* of the Jacobian, the reverse derivative computes a linear combination of the *rows* of the Jacobian.

If the gradient of a single-output function indicates in which direction its output changes fastest (and how fast that is), then the reverse derivative of a

multi-output function indicates in which direction *a particular weighted sum of its outputs* changes fastest (and how fast that is). To see this, let a vector $\vec{u} \in \mathbb{R}^n$ be given and define $g : \mathbb{R}^n \to \mathbb{R}$ by $g\ (x_1, \ldots, x_n) = u_1 \cdot x_1 + \cdots + u_n \cdot x_n$, i.e. a weighted sum of its inputs with coefficients from $\vec{u}$. The Jacobian of g is simple: $Jg\ \vec{x} = \begin{pmatrix} u_1 & \cdots & u_n \end{pmatrix} = \vec{u}^\top \in \underline{\mathbb{R}}^{1 \times n}$. By computing the gradient of $g \circ f : \mathbb{R}^m \to \mathbb{R}$:

$$\begin{aligned}
\nabla(g \circ f)\ \vec{x} &= (D(g \circ f))^\top\ \vec{x}\ \begin{pmatrix} 1 \end{pmatrix} \\
&= (J(g \circ f)\ \vec{x})^\top \cdot \begin{pmatrix} 1 \end{pmatrix} \\
&= (Jg\ (f\ \vec{x}) \cdot Jf\ \vec{x})^\top \cdot \begin{pmatrix} 1 \end{pmatrix} \\
&= (Jf\ \vec{x})^\top \cdot (Jg\ (f\ \vec{x}))^\top \cdot \begin{pmatrix} 1 \end{pmatrix} \\
&= (Df)^\top\ \vec{x}\ \left((\vec{u}^\top)^\top \cdot \begin{pmatrix} 1 \end{pmatrix}\right) \\
&= (Df)^\top\ \vec{x}\ \vec{u}
\end{aligned}$$

we see that indeed, the reverse derivative of f can be interpreted in terms of the gradient of a linear combination of f's outputs.

In practice, one very rarely needs the full generality of a reverse derivative; usually one only needs a gradient. However, as should be clear by now, computing the gradient of a function $f : \mathbb{R}^m \to \mathbb{R}$ involves just one call to $(Df)^\top$; this is much better than using the forward derivative Df, which, as we have seen at the start of this subsection, would require m calls.

Vector–Jacobian product. One can rewrite the mathematical definition of $(Df)^\top$ as follows:

$$(Df)^\top\ \vec{x}\ \vec{u} = (Jf\ \vec{x})^\top \cdot \vec{u} = (\vec{u}^\top \cdot Jf\ \vec{x})^\top$$

where the uses of $(-)^\top$ in the right-most expression are operationally uninteresting because they just convert between column and row vectors. Because of the right-most form, the reverse derivative is often called a *vector–Jacobian product*; accordingly, and analogously to "JVP" for forward AD, a common name for the reverse derivative in AD implementations is VJP.

Spoiler: reverse AD. As it turns out, using the *reverse* mode of AD, one can compute *reverse* derivatives with a constant-factor overhead over the original function. The same complexity criterion as for forward AD holds: if $f\ \vec{x}$ takes time T, then $(Df)^\top\ \vec{x}\ \vec{u}$ can be computed in time $O(T)$. (Memory usage of $(Df)^\top$ is much higher than that of f, however, and the constant factors implicit in the big-O notation are higher for reverse AD than they are for forward AD.) We will look closely at how reverse AD algorithms work in Section 2.2.4 and onwards.

2.2.3 Dual-numbers forward AD

Before we move on to implementing reverse AD, let us look at a classical implementation of forward AD first. Define a data type (using Haskell notation[19]):

$$\textbf{data}\ \text{Dual}\ a = \text{Dual}\ a\ a$$

Contrary to normal Haskell notation, however, we will write values Dual $x\ y$ suggestively as "$x + y\varepsilon$": the intuitive semantics is "x, but with an infinitesimal change y applied to it", or "x with a rate of change of y".

From the notion of an "infinitesimal change", one can intuit how these *dual numbers* should behave under arithmetic operations. If a is nudged by an amount b, and c is nudged by an amount d, how much is $a + c$ nudged? Well, by $b + d$, hence:

$$(a + b\varepsilon) + (c + d\varepsilon) = (a + c) + (b + d)\varepsilon$$

Similarly we can get:

$$(a + b\varepsilon) \cdot (c + d\varepsilon) = a \cdot c + (bc + ad)\varepsilon \tag{2.9}$$

because the nudging effect of b is scaled by c in the multiplication, and the nudging effect of d is scaled by a analogously. We ignore the "double-infinitesimal" nudge of $bd\varepsilon^2$, because as before in Section 2.2.1, (first-order) derivatives only describe a linear approximation.[20]

These equations can be computed using derivatives too, but to make this convenient, we need a slightly different partial derivative notation.

> **Notation.** So far, the notation "$\frac{\partial f}{\partial x}(v)$" required that f is a function of which one scalar (i.e. single-dimensional) argument is called x, and denoted the partial derivative of f with respect to its x argument when evaluated at the (full) input v. However, in a computation such as:
>
> $$\lambda a.\ \textbf{let}\ b = 2a\ \textbf{in let}\ c = 3b + 7a^5\ \textbf{in let}\ d = c + 1\ \textbf{in}\ d^2 + a$$
>
> it makes little sense to write something like "$\frac{\partial d}{\partial b}(?)$", as d is not a function in the first place; yet there is, in fact, a sensible way to talk about the "derivative of d with respect to b": if b changes by ε (by modifying its own assignment, not by modifying earlier variables such as a), how much does d change? Well, by 3ε. We shall write this as: $\frac{\partial d}{\partial b} = 3$. Similarly, $\frac{\partial c}{\partial a} = 3 \cdot 2 + 7 \cdot 5a^4 = 6 + 35a^4$. In this example, "$\frac{\partial b}{\partial c}$" is undefined because c in fact depends on b.

[19] This is a data type called 'Dual' with one type parameter (a); its values are normally written 'Dual $x\ y$' for values x, y of type a.

[20] We are actually computing length-2 prefixes of a Taylor series here.

Now let us return to dual numbers and consider Eq. (2.9). Assume that we have quantities a and c that depend on some initial value α with derivatives $\frac{\partial a}{\partial \alpha}$ and $\frac{\partial c}{\partial \alpha}$. Then their product depends on α as follows:

$$\frac{\partial (a \cdot c)}{\partial \alpha} = \frac{\partial a}{\partial \alpha} \cdot c + a \cdot \frac{\partial c}{\partial \alpha} \tag{2.10}$$

by the product rule for differentiation. Suppose that α changes by ε. Then a changes by $\frac{\partial a}{\partial \alpha} \cdot \varepsilon$, c changes by $\frac{\partial c}{\partial \alpha} \cdot \varepsilon$, and $a \cdot c$ changes by $\frac{\partial (a \cdot c)}{\partial \alpha} \cdot \varepsilon = \left(\frac{\partial a}{\partial \alpha} \cdot c + a \cdot \frac{\partial c}{\partial \alpha}\right)\varepsilon$ by Eq. (2.10). If we assume $b = \frac{\partial a}{\partial \alpha}$ and $d = \frac{\partial c}{\partial \alpha}$, then this matches Eq. (2.9) precisely.

This approach generalises. For an n-ary arithmetic operation $op(x_1, \ldots, x_n)$, its lifting to dual numbers is:

$$op(x_1 + y_1\varepsilon, \ldots, x_n + y_n\varepsilon) = op(x_1, \ldots, x_n) + \left(y_1 \cdot \frac{\partial op}{\partial x_1}(\vec{x}) + \cdots + y_n \cdot \frac{\partial op}{\partial x_n}(\vec{x})\right)\varepsilon$$

For example, we get also:

$$\sin(a + b\varepsilon) = \sin(a) + b\cos(a)\,\varepsilon \qquad \text{since } \tfrac{\partial \sin}{\partial x}(x) = \cos(x)$$

$$\sqrt{a + b\varepsilon} = \sqrt{a} + \tfrac{b}{2\sqrt{a}}\,\varepsilon \qquad \text{since } \tfrac{\partial \sqrt{-}}{\partial x}(x) = \tfrac{1}{2\sqrt{x}}$$

Forward derivative. If one has a function $f : \mathbb{R}^m \to \mathbb{R}^n$ that acts on its inputs only using operations with well-defined partial derivatives (hence, operations that can be lifted to dual numbers), then we can *reinterpret* the function f as having type $(\text{Dual } \mathbb{R})^m \to (\text{Dual } \mathbb{R})^n$. Identifying Dual $\mathbb{R}$ and $\mathbb{R} \times \underline{\mathbb{R}}$, we get $(\mathbb{R} \times \underline{\mathbb{R}})^m \to (\mathbb{R} \times \underline{\mathbb{R}})^n$; zipping the arguments and unzipping the results allows us to present an external interface of type $\mathbb{R}^m \times \underline{\mathbb{R}}^m \to \mathbb{R}^n \times \underline{\mathbb{R}}^n$, which takes values and perturbations and returns values and perturbations. Dropping the first half of the output (leaving just the output perturbations) and uncurrying yields $\mathbb{R}^m \to \underline{\mathbb{R}}^m \to \underline{\mathbb{R}}^n$, which is the expected type for Df, and the constructed function is in fact a valid implementation of Df.

Implementation. Constructing the forward derivative using dual numbers in a programming language is straightforward. In Fig. 2.5, we give a small implementation in Haskell for illustration. Note that in Fig. 2.5 we implement only the `Num` class (interface) for conciseness; other classes like `Fractional` and `Floating` could be implemented just as easily, as long as one has (partial) derivatives for the individual operations. If this is done, the constraint list on `b` in the first argument to `forwardDer` can be extended accordingly.

2.2.4 Data flow graphs

Consider a very simple programming language with only one type ($\mathbb{R}$), and with terms consisting of constants, arithmetic operations and some form of sharing

```haskell
data Dual a = Dual a a

instance Num a => Num (Dual a) where
  Dual x dx + Dual y dy = Dual (x + y) (dx + dy)
  Dual x dx - Dual y dy = Dual (x - y) (dx - dy)
  Dual x dx * Dual y dy = Dual (x * y) (dx*y + x*dy)
  abs (Dual x dx) = Dual (abs x) (dx * signum x)
  signum (Dual x _) = Dual (signum x) 0
  fromInteger n = Dual (fromInteger n) 0

forwardDer :: Num a
           => (forall b. Num b => [b] -> [b])
           -> [a] -> [a] -> [a]
forwardDer f inputs tangents =
  map (\(Dual _ dy) -> dy) (f (zipWith Dual inputs tangents))
```

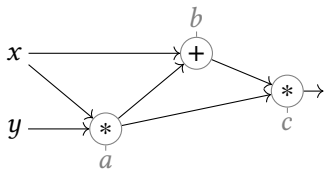
Figure 2.5: A tiny implementation of dual-numbers forward AD in Haskell. The call to `f` in `forwardDer`'s body instantiates its type variable `b` to `Dual a`.

(e.g. let-bindings). Programs have zero or more input values, and one or more output values. A traditional syntax for this language could be:[21]

$$\begin{aligned} \mathit{program} &::= \lambda x_1 \ldots x_n.\ \mathit{body} \\ \mathit{body} &::= \textbf{let } x_i = \mathit{rhs} \textbf{ in } \mathit{body} \mid (x_{i_1}, \ldots, x_{i_n}) \\ \mathit{rhs} &::= r \mid -x_i \mid x_i + x_j \mid x_i * x_j \mid \sin x_i \mid \cos x_i \mid \ldots \end{aligned} \tag{2.11}$$

where "r" stands for constants of type $\mathbb{R}$, such as '2.7'; the '...' on the last line stands for any further arithmetic operations that one wants to add, as long as they take variables as inputs and return a single output scalar. This language is essentially *administrative normal form* (ANF) without any control flow and only arithmetic operations as expression constructors. In imperative programming, such programs are often called *straight-line programs* (there being no branching).

The intended syntax for this language, however, is not traditional terms but instead *data flow graphs*. Here is an example data flow graph:



(2.12)

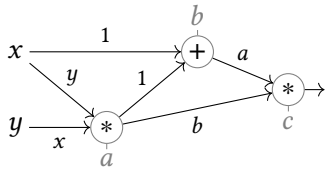
This graph has two inputs (x, y) and one output, and corresponds to the program

$$\lambda x\, y.\ \textbf{let } a = x * y \textbf{ in let } b = x + a \textbf{ in let } c = b * a \textbf{ in } (c) \tag{2.13}$$

[21] We use $(*)$ instead of $(\cdot)$ for scalar multiplication for readability in the graphs.

in the syntax of Eq. (2.11). Note that not only the inputs, but also all intermediate nodes of the graph (including the output, c), have names. Data flow graphs must be acyclic, but they may be non-simple if an operation uses the same input twice (such as in $\lambda x.\ \mathbf{let}\ y = x * x\ \mathbf{in}\ (y)$).

Because we are interested in differentiation, let us annotate the partial derivatives of each arithmetic operation on the corresponding arrow in Eq. (2.12):



(2.14)

For example, we have $\frac{\partial c}{\partial a} = \frac{\partial (a*b)}{\partial a} = b$, hence the derivative annotated on the arrow $a \longrightarrow c$ is 'b'.

Chain rule. The chain rule from calculus is typically presented for two functions $f, g : \mathbb{R} \longrightarrow \mathbb{R}$:

$$\frac{\mathrm{d}f(g(x))}{\mathrm{d}x} = \frac{\mathrm{d}f(u)}{\mathrm{d}u}(g(x)) \cdot \frac{\mathrm{d}g(u)}{\mathrm{d}u}(x)$$

where the 'd', as opposed to '∂', is to stress that these are derivatives with respect to the *only* arguments of these functions. In data flow graphs, this says that for the following function:

we have $\frac{\partial b}{\partial x} = d_1 \cdot d_2$. This is rather unsurprising: if changing x by ε makes a change by $d_1\varepsilon$, and changing a by δ makes b change by $d_2\delta$, then surely changing x by ε makes b change by $d_1 d_2 \varepsilon$ (simply set $\delta = d_1\varepsilon$). Written using forward derivatives, this becomes:

$$D(g \circ f)\ x\ u = Dg\ (f\ x)\ (Df\ x\ u) \tag{2.15}$$

There are various generalisations of the chain rule in common use, but they are often rather heavy on indices, and still usually handle only compositions of two functions (albeit of multiple variables). In terms of these data flow graphs, however, we can more easily give a very general version of the chain rule.[22]

[22]This version also appeared in Eq. 26 of [Linnainmaa 1976].

Theorem 1 (Chain rule)**.** *Given a data flow graph G where all nodes are differentiable, and given two nodes s, t in G (the source s is allowed to be an input) such that there is a path from s to the target t, we have:*

$$\frac{\partial t}{\partial s} = \sum_{\substack{\text{paths } p \text{ from} \\ s \text{ to } t}} \prod_{\substack{\text{edges } e \\ \text{in } p}} (\textit{partial derivative on } e).$$

The proof of this theorem is by induction on the graph in topological order, applying a more usual version of the chain rule at each visited node; the details are out of scope here. However, we can build some confidence in its correctness by looking back at the example graph in Eq. (2.14). In this graph, there are three paths from x to c ($x \xrightarrow{1} b \xrightarrow{a} c$; $x \xrightarrow{y} a \xrightarrow{1} b \xrightarrow{a} c$; $x \xrightarrow{y} a \xrightarrow{b} c$) and two paths from y to c ($y \xrightarrow{x} a \xrightarrow{1} b \xrightarrow{a} c$; $y \xrightarrow{x} a \xrightarrow{b} c$); thus, applying Theorem 1, we get the following partial derivatives:

$$\frac{\partial c}{\partial x} = 1 \cdot a + y \cdot 1 \cdot a + y \cdot b \qquad \frac{\partial c}{\partial y} = x \cdot 1 \cdot a + x \cdot b$$

Some algebra, and the fact that $a = xy$ and $b = x + xy$, verifies that these are indeed the two partial derivatives of the function defined by Eq. (2.13).

2.2.5 Forward AD on data flow graphs

Directly computing the sum of products in Theorem 1 is naturally extremely inefficient, because many paths share significant overlap — indeed, it can be exponential in the size of the graph. However, its computation can be significantly optimised by dynamic programming (memoisation). The trick is to not compute $\frac{\partial t}{\partial s}$ by applying the theorem directly, but to incrementally compute $\frac{\partial n}{\partial s}$ for all intermediate nodes n:

1. Filter away all nodes that are not reachable from the source node s or that cannot reach the target node t, leaving the part of the graph "suspended" between s and t. For example, in the example graph Eq. (2.14): if our goal is computing $\frac{\partial c}{\partial x}$, the only node to be removed is y.

2. Set the accumulated derivative at the source node s to 1. (After all, $\frac{\partial s}{\partial s} = 1$.)

3. Loop over all nodes in the remaining graph (except the source node) in topological order. For each visited node n, compute $\frac{\partial n}{\partial s} = \sum_p \frac{\partial n}{\partial p} \cdot \frac{\partial p}{\partial s}$, where p ranges over the predecessors[23] of n, $\frac{\partial n}{\partial p}$ is the value on the edge $p \to n$, and $\frac{\partial p}{\partial s}$ was already computed in an earlier iteration of the loop.

[23] We mean here the p for which an edge $p \to n$ exists.

Because t is in the remaining graph after step 1, in particular $\frac{\partial t}{\partial s}$ is also computed in step 3, yielding the answer.

Note that while these edge derivatives $\frac{\partial n}{\partial p}$ refer to intermediate values of the program (e.g. in Eq. (2.14), $\frac{\partial c}{\partial a}$ refers to the value of node b), these intermediate values are always predecessors of n. For this reason, the edge derivatives do *not* all need to be computed and stored in a separate phase before running this differentiation algorithm; instead, they can be computed simultaneously in the same topological order, and they will be available precisely when required. Furthermore, after all successors of a node have been visited, the edge derivatives can be forgotten: this is precisely the same moment when the intermediate values themselves could be forgotten in an evaluation of the original data flow graph. Hence, the memory usage of this algorithm is proportional to the memory usage of evaluation of the original graph.

We can generalise this algorithm in two ways:

1. We can compute partial derivatives of *multiple* target nodes with respect to the same source node in one traversal of the graph: in fact, the algorithm already computes *all* partial derivatives with respect to the source node that are included in the reduced graph after step 1, so we only need to filter away fewer nodes.

2. We can allow multiple source nodes instead of just one, provided that a suitable initialisation is given for each of them for step 2 of the algorithm.

When given source nodes $s_1, \ldots, s_m$ with values $x_1, \ldots, x_m$ and derivative initialisers (tangents) $u_1, \ldots, u_m$ (for step 2) as in the second generalisation, and target nodes $t_1, \ldots, t_n$ as in the first generalisation, this algorithm will compute:

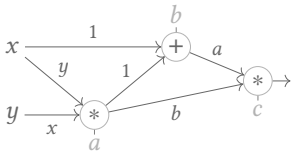
$$\sum_{i=1}^{m} u_1 \cdot \left(\frac{\partial t_1}{\partial s_i}, \ldots, \frac{\partial t_n}{\partial s_i}\right) = Jf\ \vec{x} \cdot \vec{u} = Df\ \vec{x}\ \vec{u} \tag{2.16}$$

where $f : \mathbb{R}^m \to \mathbb{R}^n$ is the implied function that computes $t_1, \ldots, t_n$ from the inputs $s_1, \ldots, s_m$.

This algorithm is the essence of forward AD.

If we execute this graph-based forward AD algorithm on our example graph in Eq. (2.14), we start with inputs x, y and input tangents dx, dy, and successively compute: (on the right is a reminder of Eq. (2.14))

- $a = x * y;\ \ da = y * dx + x * dy$
- $b = x + a;\ \ db = dx + da$
- $c = b * a;\ \ dc = a * db + b * da$



As we can see, this does not only produce the forward derivative at the given inputs and tangents: it also computes the original function result alongside it. This is common to all forward AD algorithms, and implementations will typically not discard the computed function result but return it to the user together with the derivative. The computed original-function intermediate values (including the output(s); here: a, b, c) are called the *primals*; the computed derivatives (here: da, db, dc) are called *duals* or *tangents*. We say that forward AD *interleaves* the *primal computation* and the *dual computation*.

We can now understand the dual-numbers algorithm from Section 2.2.3 as a particular instantiation of this algorithm: where the original program computed a, b, c, the program transformed using dual-numbers forward AD would compute pairs $a + da\,\varepsilon$, $b + db\,\varepsilon$, $c + dc\,\varepsilon$.

Complexity of forward AD. We assume that the filtered graph of step 1 is already given; this is in practice accomplished by the compiler eliminating dead code (i.e. removing nodes that cannot reach the target nodes) and lifting constant values out of a lambda function (i.e. removing nodes not reachable from the source nodes).[24] Step 2 performs $O(\#S)$ work, where $\#S$ is the number of source nodes.

Then, the core loop of the algorithm (step 3) visits every node exactly once. In each node, it performs the original operation and additionally computes the forward derivative of this operation. If the forward derivative of all primitive operations can be computed in time proportional to the operations themselves (trivially true for constant-time arithmetic operations with a constant-time derivative), then the runtime of step 3 is proportional to the combined runtime of the nodes in the graph. Writing T_{graph} for this combined runtime, we thus get that forward AD computes the forward derivative of a data flow graph in time $O(T_{\text{graph}} + \#S)$.

In practice, we have a *program*, not a graph, but as we have seen in Section 2.2.3, there is no need to build an explicit graph: computing the original program result and its forward derivative in tandem, we only add computation for the differentiable operations, leaving the rest of the program alone. Writing T for the runtime of the full original program, I for the size of its full input and O for the size of its full output, we need to select the scalars from the input ($O(I)$), initialise the source nodes ($O(\#S) \subseteq O(I)$), compute the primals ($O(T)$) and the tangents ($O(T_{\text{graph}}) \subseteq O(T)$), and finally report the outcoming tangents ($O(O) \subseteq O(T)$), summing to $O(T + I)$, meaning that the forward derivative of a *program* can be computed in time $O(T + I)$.

If we proceed make the rather reasonable assumption that most of the pro-

[24] In a language with dynamic control flow, a compiler may not be able to filter away all redundant computation. In this case, the analysis here is on the graph built by tracing execution of the program that is actually run; this trace graph may have redundant nodes, but no more than the user wrote.

gram's inputs are actually used, we get $I \in O(T)$ and thus finally $O(T + I) = O(T)$. That is to say: the forward derivative can be computed with at most a constant factor overhead in runtime over the original program (and there is a bound on this constant that is independent of the program being differentiated).

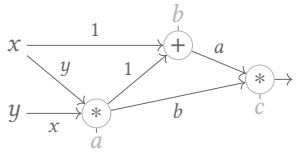
The memory use of the forward derivative computation is also proportional to that of the original function, as every scalar is simply replaced by two scalars, and these scalars retain their original lifetime.

2.2.6 Reverse AD on data flow graphs

The product in Theorem 1 does not prescribe any particular order, so it is not very strange to consider changing the forward AD algorithm on data flow graphs to accumulate in *reverse topological order*, instead of the usual order. In this way, we cannot compute node values and their accumulated derivatives simultaneously any more; we have to first compute, and store, the (primal) value of all nodes in the graph in the usual, forward evaluation direction. Afterwards, we can accumulate the products along the paths in the reverse direction.[25] The "source nodes" for this reverse traversal are the original target nodes, and the "target nodes" for this reverse traversal are the original source nodes.

Looking again at Eq. (2.14) and executing this reverse algorithm on that graph, we get the following computation. (A reminder of the graph is shown on the right, as before.) The inputs are x, y as well as dc, the initialisation of the derivative at the (in this case single) output c.

- $a = x * y$
- $b = x + a$
- $c = b * a$
- $db = a * dc$
- $da = b * dc + 1 * db$
- $dx = 1 * db + y * da$
- $dy = x * da$



Where the forward algorithm computed $\frac{\partial n}{\partial s}$ for each node n in the graph (and source node s), and possibly a linear combination of such partial derivatives in the case of multiple source nodes, this reverse algorithm can be seen to compute $\frac{\partial t}{\partial n}$ for each node n in the graph (and target node t). It computes a linear combination of such partial derivatives when given multiple *target* nodes. Where extending

[25] We can choose whether to compute the partial derivatives on the arrows either in the forward pass or in the reverse (accumulation) pass. If we choose the former, the primal values need not be stored for the reverse pass; if we choose the latter, the maximum memory usage is a little lower.

the forward algorithm to multiple target nodes was trivial (just enlarge the graph to traverse), the reverse algorithm trivially extends to multiple *source* nodes.

Summarising: when given source nodes $s_1, \ldots, s_m$ with values $x_1, \ldots, x_m$, and target nodes $t_1, \ldots, t_n$ with their derivative initialisers (*cotangents*) $u_1, \ldots, u_n$, the reverse algorithm computes the reverse derivative:

$$\sum_{i=1}^{n} u_i \cdot \left(\tfrac{\partial t_i}{\partial s_1}, \ldots, \tfrac{\partial t_i}{\partial s_m}\right) = \sum_{i=1}^{n} u_i \cdot \nabla f_i \, \vec{x} = (Df)^\top \, \vec{x} \, \vec{u}$$

where $f : \mathbb{R}^m \to \mathbb{R}^n$ (with $f_i \coloneqq (\vec{x} \mapsto (f\,\vec{x})_i)$) is again the implied function that computes $t_1, \ldots, t_n$ from the inputs $s_1, \ldots, s_m$. Compare this with Eq. (2.16) for the forward algorithm. (For the last equality here, it may be helpful to review Eq. (2.8) that writes $(Df)^\top$ in terms of ∇f_i.)

This "reverse accumulation" algorithm is the essence of reverse AD.

As before with forward AD, we see that we do not only compute the reverse derivative: the computation also happens to produce the normal function result, just like for forward AD. And like before, reverse AD implementations typically do not discard this function result, but simply return it alongside the reverse derivative. We call the computed a, b, c again *primals*; the computed derivatives db, da, dx, dy are called *duals* or *cotangents*.[26]

Complexity of reverse AD. Unlike with forward AD, the primal and dual computations are *not* interleaved — and in fact they cannot be, because the primals are necessarily computed in the forward direction, but every step of the reverse pass over the graph immediately needs the primals of the inputs to the visited node. The only way to make those primals available is to run the forward pass to completion first, storing all (or most of) the intermediate results, and *then* running the reverse pass. This shows how in general, the memory use of a program differentiated using reverse AD is proportional to the *runtime* of the original program.[27] This rather large increase in memory consumption over the original program can be traded off against additional computation time using a technique called *checkpointing* (see Section 2.2.9 below).

The time complexity of the reverse algorithm is similar to that of the forward algorithm, except the roles of the input size and the output size are swapped; nevertheless, we still get $O(T + O + I) = O(T + I + O) = O(T)$ under the mild

[26]The terminology of 'tangent' (for forward AD) versus 'cotangent' (for reverse AD) comes from differential geometry; see also Section 5.2.

[27]It is proportional to the combined storage requirements for all graph nodes; if we make the graph nodes elementary enough, this is the same as $O(\#(\text{graph nodes}))$, and if we further assume that most of the program actually does differentiable computation, this is well approximated by $O(\text{original program runtime})$.

assumption that most of the input is actually used by the program. This result is a classical one (hinted at by Wolfe [1982], proved by Baur and Strassen [1983] and now known as the Baur–Strassen theorem or the *cheap gradient principle*; see also [Griewank and Walther 2008]), and is the (algorithmic) foundation for why e.g. modern machine learning is computationally feasible.[28]

In this thesis, this complexity result will show up not as a given but as a *goal*: because we know it is possible, algorithms that do not attain it should be improved until they do (or nearly do). For example, most of Chapter 3 concerns taking a (very naive but semantically elegant) reverse AD code transformation where the differentiated program takes time closer to $O(2^T)$, and optimising it via $O(T(I + \log(T)))$ and $O(T(\log(I) + \log(T)))$ to finally the desired $O(T)$ in a principled fashion. In practice, the complexity analyses in Chapters 3 and 6 use the more precise $O(T + I)$ instead of $O(T)$, because it tends to be more convenient in the arguments and to avoid making unnecessary assumptions.

Reverse AD on program terms. The language of data flow graphs is hardly attractive to program in. Practical programming languages support data structures beyond just scalars, and admit dynamic control flow that determines which computations will actually be performed based on runtime values. For example, consider the following program in a somewhat extended language:

$$\lambda a.\ \textbf{let}\ b = a + 1\ \textbf{in if}\ b > 0\ \textbf{then}\ a * b\ \textbf{else}\ b \tag{2.17}$$

There is no one data flow graph equivalent to this program, as it would depend on whether the condition $b > 0$ is true at runtime or not:

As we have seen, the lack of an explicit data flow graph is not a problem for forward AD: the dual-numbers algorithm of Section 2.2.3 has no need for such a graph. For reverse AD, however, things are not so simple. One solution to the lack of an explicit graph is to simply construct one: instrument the program to be differentiated so that it not only computes its normal output, but also builds up the data flow graph for this particular execution of the program at runtime. In the reverse pass, this graph is then traversed following the graph reverse AD algorithm discussed in this section. This approach is called *taping*, *tracing* or sometimes *define-by-run* reverse AD, and is discussed in more detail in Section 2.2.8.

[28]The other foundation is spectacular hardware improvements in computations per Watt.

However, explicitly constructing the graph at runtime can have undesirable overhead, and as such one can try to do more work at compile time. Given a term (program) t to differentiate, we can try to construct a term t' that "specialises" the graph reverse AD algorithm: evaluating t' on an input x (and some initial cotangents d) should perform the same computations as the graph reverse AD algorithm would have done on the data flow graph for t on x. Of course, the point is that t' should not actually construct (most of) that graph. For example, if t is the term in Eq. (2.17), a suitable t' could be:

$$\lambda a\, d.\ \textbf{let}\ b = a + 1\ \textbf{in if}\ b > 0\ \textbf{then}\ b * d + a * d\ \textbf{else}\ d$$

If one writes out the computations done by the graph-based reverse pass on both versions of the graph of Eq. (2.17), like we did on page 31 for the original example graph Eq. (2.12), one sees that this t' indeed performs the same arithmetic computation — we just inlined the bindings that were used only once anyway to simplify the resulting term.

Reverse AD algorithms that avoid constructing a graph in this way at runtime are sometimes called *define-then-run* reverse AD. Most of the approaches discussed in the remainder of this chapter are of this form, as well as the second approach to reverse AD studied in this thesis (CHAD, Chapters 5 to 7). The first approach studied in this thesis (dual-numbers reverse AD, Chapter 3) is by nature a taping-style algorithm, but in the process of optimising it (Chapter 4), it turns into a define-then-run algorithm as well.

Reverse derivative of sharing and dropping. When the discussed forward and reverse graph differentiation algorithms visit a node that, from their perspective, has multiple incoming edges (i.e. multiple *incoming* edges for forward AD, but multiple *outgoing* edges for reverse AD), the contributions along each edge are added together. This resulted in the uses of $(+)$ in the assignments to the tangent variables *da*, *db*, *dc* on page 29 and to some of the cotangent variables (*da*, *dx*) on page 31.

In the world of terms, a node having multiple outgoing edges means that a value is *shared* (stored in a variable and subsequently referenced multiple times). Thus, in reverse AD, we observe that sharing in the original program translates to addition in the dual. In a similar fashion, if a value is actually *unused* in the original program (i.e. its node in the data flow graph has zero outgoing edges), its cotangent in reverse AD ought to be zero; we can formulate this as: dropping a value in the original program translates to zero in the dual.

These observations make sense from the perspective of derivatives too. We can represent duplication (the elementary form of sharing) by the function $f_{\text{dup}} = \lambda x.\ (x, x) : \mathbb{R}^1 \to \mathbb{R}^2$; in the same sense, the function representing dropping is

$f_{\text{drop}} = \lambda x.\ () : \mathbb{R}^1 \to \mathbb{R}^0$. These are their Jacobian matrices:

$$Jf_{\text{dup}}\ x = \begin{pmatrix} 1 \\ 1 \end{pmatrix} \in \underline{\mathbb{R}}^{2\times 1} \qquad Jf_{\text{drop}}\ x = \begin{pmatrix} \end{pmatrix} \in \underline{\mathbb{R}}^{0\times 1}$$

Hence, their reverse derivatives are the following:

$$(Df_{\text{dup}})^\top\ x\ (u_1, u_2) = (Jf_{\text{dup}}\ x)^\top \cdot \begin{pmatrix} u_1 \\ u_2 \end{pmatrix} = \begin{pmatrix} 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} u_1 \\ u_2 \end{pmatrix} = u_1 + u_2$$
$$(Df_{\text{drop}})^\top\ x\ () = (Jf_{\text{drop}}\ x)^\top \cdot \begin{pmatrix} \end{pmatrix} = \begin{pmatrix} \end{pmatrix} \cdot \begin{pmatrix} \end{pmatrix} = 0$$

Note that despite the confusing notation, the second derivation is, in fact, type-correct: the "$\begin{pmatrix} \end{pmatrix} \cdot \begin{pmatrix} \end{pmatrix}$" multiplies a 1×0 matrix with a length-0 column vector, producing a length-1 column vector as a result. (As usual, we identify $\underline{\mathbb{R}}^1$ and $\underline{\mathbb{R}}$ in the results above.) If the reader is unconvinced, they may recall that $(Df_{\text{drop}})^\top\ x$ is a *linear* function from $\underline{\mathbb{R}}^0$ to $\underline{\mathbb{R}}^1$; because a linear function sends 0 to 0, and the only element in $\underline{\mathbb{R}}^0$ *is* 0, there is exactly one such function: $\lambda().\ 0$.

The "sharing becomes addition" and "dropping becomes zero" observations are natural in the context of the graph reverse AD algorithm, but they result in some important design trade-offs when designing reverse AD algorithms on terms. Indeed, sharing and dropping are normally *implicit* operations: for example, one does not typically indicate sharing with an explicit duplication combinator, outside of resource-linear type systems (see e.g. [Wadler 1990], as well as the Rust, Clean [Brus et al. 1987; de Vries et al. 2007] and Futhark programming languages, among others). Sharing and dropping are also $O(1)$ operations, whereas addition and zero are not. Furthermore, sharing of variables in the source program does not necessarily result in multiple outgoing edges in the data flow graph; for example, in Eq. (2.17), a is definitely shared syntactically, but has only one outgoing edge in the graph if $b \leq 0$. The issues described in this paragraph will return later in this thesis when looking at concrete algorithms.

2.2.7 Data flow graphs for SOACs

When designing a reverse AD algorithm for a particular language, one not only needs to decide how to handle variable binding and control flow constructs, but one also needs derivatives of all of the primitive operations in the language. For arithmetic operations like (+), elementwise multiplication of arrays, or other first-order operations, this is relatively easy and unrelated to AD specifically: one either looks up the derivative in a table (e.g. to remember that $\frac{\partial \sin^{-1}(-)}{\partial x}(x) = \frac{1}{\sqrt{1-x^2}}$) or derives it using standard methods from calculus.

Higher-order primitive operations, however, need a little more thought. Let us look at what is arguably the simplest non-trivial second-order array combinator:[29]

$$\text{map} : (a \to b) \to \text{Array}\ a \to \text{Array}\ b$$

Suppose that we are designing a reverse AD algorithm and we need to be able to differentiate expressions 'map $f\ x$'. Assume that we are in a purely functional language and that the function f is closed (i.e. has no free variables). How do we approach this?

The first thing to observe is that the computation of a reverse derivative of a function f by reverse AD consists of two phases:

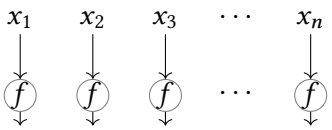
- The forward pass: take an input and evaluate f on it in the forward direction, producing the normal function result as well as the computed intermediate values;
- The reverse pass: take the computed intermediate values as well as a cotangent for f's output, and produce cotangents for the inputs of f.

For an $f : a \to b$, we can thus say that reverse AD produces two functions:

$$f_{\text{fwd}} : a \to (b, c) \qquad f_{\text{rev}} : c \to \underline{b} \to \underline{a}$$

where c is some additional type produced by the reverse AD algorithm that depends on the implementation of f, and records f's intermediate values for the reverse pass; the underlined types $\underline{a}$ and $\underline{b}$ denote the types of cotangents, and can be taken equal to the original types a and b for simplicity. In the derivation on page 31, the type c would have been $\mathbb{R}^2$: a pair of the scalars a and b, as c, being the function output, was unnecessary for the reverse pass.

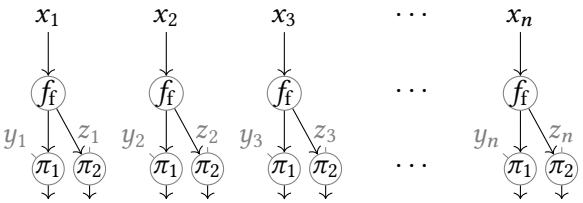
Differentiating 'map'. As an example, let us consider the array x an input, and construct a reverse derivative for the function $g\ x = \text{map}\ f\ x$. We are given, by a recursive application of reverse AD, f_{fwd} and f_{rev}, and we have to produce g_{fwd} and g_{rev}. Let us draw the data flow graph of the original function g first:



[29] Support for multi-dimensional arrays is orthogonal to the topic at hand, so as before, all our arrays here are single-dimensional.

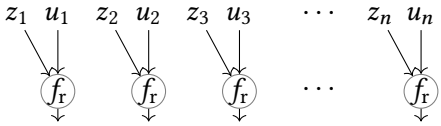
writing the elements of the input x as $x_1, \ldots, x_n$, and collecting the outputs of this graph as the output array. Notable is that individual nodes in this graph are no longer necessarily scalars; the x_i nodes are of type a and the f nodes are of type b.

To implement g_{fwd}, we simply replace the uses of f by f_{fwd} (abbreviated as 'f_{f}' in the graph below). We use π_1 (the first projection from a tuple, also written 'fst') and π_2 (the second, i.e. 'snd') to produce the desired results from the output of f_{fwd}.



The y_i outputs should be considered collected in an array, as should the z_i outputs; if $f_{\text{fwd}} : a \to (b, c)$, then $g_{\text{fwd}} : \text{Array } a \to (\text{Array } b, \text{Array } c)$. This means that we have chosen 'Array c' as our "additional type".

For the reverse pass, we now have to implement $g_{\text{rev}} : \text{Array } c \to \text{Array } \underline{b} \to \text{Array } \underline{a}$. As a data flow graph, this is straightforward in terms of $f_{\text{rev}} : c \to \underline{b} \to \underline{a}$ (abbreviated as 'f_{r}'):



Thus, the following definitions suffice:

$$g_{\text{fwd}} : \text{Array } a \to (\text{Array } b, \text{Array } c) \qquad g_{\text{rev}} : \text{Array } c \to \text{Array } \underline{b} \to \text{Array } \underline{a}$$
$$g_{\text{fwd}}\ x = \text{unzip}\ (\text{map}\ f_{\text{fwd}}\ x) \qquad g_{\text{rev}}\ z\ u = \text{zipWith}\ f_{\text{rev}}\ z\ u$$

where $\text{unzip} : \text{Array } (a, b) \to (\text{Array } a, \text{Array } b)$ returns two arrays with the first, respectively second, components of the pairs in the input array. While there is no fully systematic way of coming up with such "nice" forms of g_{fwd} and g_{rev}, human pattern-matching on the graph forms works extremely well in practice.

If g was the entire input program to differentiate, one can construct its reverse derivative out of these two functions:

$$(Dg)^\top\ x\ u = \textbf{let}\ (\_, z) = g_{\text{fwd}}\ x\ \textbf{in}\ g_{\text{rev}}\ z\ u$$

Note that we ignore the primal output of the forward pass here, at the top level; this is typical: intermediate values are, in general, required in the reverse pass, but the final program result is not.[30]

[30]Other intermediate values that are not required in the reverse pass are arguments to linear

2.2.8 Reverse AD via taping

So far, we have looked at AD algorithms from a high-level perspective: we know what they should do to the data flow graph of a program, and in Section 2.2.7 we have seen a technique for differentiating combinators that have non-trivial subprograms. In the remainder of this chapter, we continue to focus on reverse AD (instead of forward AD), and discuss various ways to apply these ideas in practical algorithms, including optimisations that one can subsequently apply to those algorithms. Since we cannot possibly cover all interesting literature on AD, for further reading we recommend the surveys [Baydin et al. 2017; Margossian 2019] and the classical handbook by Griewank and Walther [2008]. The individual chapters of this thesis also contain more detailed literature overviews related to the specific topic at hand; see Sections 3.13, 4.9 and 6.9.

Origins of AD. One of the earliest appearances of an AD-like algorithm in the literature is [Wengert 1964], which describes forward AD by writing the function to be differentiated as a list of primitive operations (i.e. a topological order of its data flow graph, now known as a *Wengert list* or *tape*), and differentiating them each in turn; correctness follows from the chain rule. Joss [1976] proceeds with a code transformation on Algol (without conditional branches, but including loops) that computes gradients using forward AD in vectorised style: instead of carrying along tangents with respect to one input value as in dual-numbers forward AD, carry along a *vector* of tangents with respect to each individual input value. If the original function runs in time T and space S, forward AD requires time $O(T)$ and space $O(S)$, and vectorised forward AD (computing a gradient) requires time $O(nT)$ and space $O(nS)$, where n is the number of top-level program input scalars.

Linnainmaa [1976] may be the first to publish a proper reverse AD algorithm, computing a gradient in time $O(T)$ and space $O(T)$, for what is essentially a data flow graph as in Section 2.2.4. Independently, Speelpenning [1980] gives a code transformation with a similar design on Fortran programs without conditional branches.[31] Both algorithms are taping algorithms, but elect to store not the operations executed in the forward pass with their primal results, but rather the operations' partial derivatives with respect to their arguments. In the language of data flow graphs from Section 2.2.6, what is stored are the partial derivatives on the edges, rather than the nodes and their values. This simplifies the reverse pass.

More recent implementations that use a pure taping approach include Autograd [Maclaurin 2016] and the traditional implementation of PyTorch [Paszke et al. 2017] (before its XLA backend) in machine learning, and Stan Math [Carpenter

operations such as (+), as their partial derivatives do not mention their primal arguments.

[31] The space complexity in [Speelpenning 1980, §7] is reported as $O(S)$, but this is because they count only RAM usage, ignoring disk storage used for the primals (§4.5).

et al. 2015] in probabilistic programming. All of these implementations generalise the tape beyond just scalars, allowing bulk array operations to be a single entry on the tape, thereby significantly reducing the runtime overhead of taping.

Taping, indirectly. There are various techniques from functional programming that allow one to write reverse AD algorithms in a more compositional style, while nevertheless having the same operational behaviour as a taping implementation. One such approach is dual-numbers reverse AD, which is the subject of Chapter 3. Other approaches use non-standard control operators like delimited continuations (e.g. [Wang et al. 2018]) or effect handlers (see [Sigal 2024]); both of these techniques achieve an effect similar to CPS-transforming[32] the user program, except that the continuation calls need not necessarily be tail calls. The result is that the transformed program creates a list of all executed primitive operations on the *call stack*; the associated AD methods reuse this automatic "tape" on the call stack as the tape for reverse AD. More details are given in Section 3.13.2.

Symbolic taping / tracing AD. Differentiating only at runtime, when a tape has been collected and all structure in the program has been evaluated away, has advantages: the reverse AD algorithm need not concern itself with whatever data structures and control flow constructs the source program used, and can simply differentiate a straight-line program in a small language. The downside, naturally, is needing to collect this tape again for every evaluation of the derivative at a new input value.

To address this problem, one can evaluate away only *some* of the structure in the source program using *tracing* (symbolic execution into a simpler language). In this approach, one partially evaluates the source program on symbolic inputs, and collects a trace of the operations it ended up executing on those inputs. Control flow (transitively) dependent on the inputs must be represented in the symbolic trace, whereas control flow that depends only on constants or hyperparameters is evaluated away. The collected trace (in a smaller, simpler language) is then differentiated using define-then-run methods (discussed in Section 2.2.10) and compiled. Reverse AD implementations that work this way include PyTorch, JAX [Bradbury et al. 2018], Dr.Jit [Jakob et al. 2022], and others.

2.2.9 Optimisations on taping reverse AD

Reducing memory use. The most important downside of reverse AD over forward AD is its memory usage: the derivative program has peak memory usage

[32] Continuation-passing style, where function results are "returned" by calling a continuation instead of returning them directly. See e.g. [Reynolds 1993].

$O(T)$, where T is the runtime of the original program; this can be problematic because memory is relatively scarce, especially on discrete GPU hardware. A technique that addresses this problem is *checkpointing*, which reduces peak memory usage at the cost of an increase in computation. Noting that the large memory consumer in reverse AD is the storage of primal values from the forward pass until they are required in the reverse pass, the idea of checkpointing is to skip storing some primal values in the forward pass. Each time missing primals are encountered in the reverse pass, they are recomputed by repeating a the corresponding part of the forward pass.[33] With a clever checkpointing scheme of Siskind and Pearlmutter [2018], one can even asymptotically reduce memory overhead to $O(\log T)$, at the cost of a derivative runtime of $O(T \log T)$ instead of the expected $O(T)$. An early reference on checkpointing is [Dauvergne and Hascoët 2006]; a good explanation and further references can be found in [Margossian 2019, §3.2.3].

Partial structure preservation. In general, the idea behind the taping approach to reverse AD is to remove structure from the source program until it becomes trivial to reverse-differentiate. However, retaining some structure can be beneficial for performance. The Adept AD library [Hogan 2014] uses a C++ technique called *expression templates*[34] to be able to statically differentiate fragments of the source program between instances of dynamic control flow, getting the benefits of define-then-run reverse AD (Section 2.2.10) on the simple parts of the source program while retaining the flexibility of taping AD globally.

Orthogonally, one can also observe that while loops are common in imperative programs, actual *dynamic* control flow (i.e. branching based on conditions computed from input values) is uncommon in some uses of reverse AD, such as traditional neural networks. In this situation, the runtime-collected tape may usually, or always, have the same structure, and one may benefit from allocating it only once with the "common" structure. This technique is called *retaping* (see [Margossian 2019, §3.2.2]).

2.2.10 Define-then-run reverse AD

All reverse AD algorithms somehow collect primals in some data structure that can, with more or less imagination, be seen as a tape. In the case of taping-style reverse AD that we discussed before (also called *define-by-run* reverse AD), the tape is explicit and linear and contains not only the primal values, but also which (arithmetic) operations produced those primal values. The reverse pass

[33] This only works properly if the program is side-effect-free.

[34] First described in C++ by Veldhuizen [1995].

then consists of an *interpreter* for the little straight-line language that the tape is written in; this interpreter interprets each operation as its transposed derivative.

Algorithms (code transformations) for reverse AD that retain most of the program structure, storing only the primal values themselves, naturally expose more structure of the derivative program to the compiler. For such algorithms, the tape is no longer linear: instead, it follows the (lexical) program structure, e.g. storing the primals from the branches of a conditional in a coproduct (sum type) to simultaneously record the control flow branch taken and the primals for that branch. We see this happening explicitly in e.g. Section 7.3.2.

The primary benefit of these algorithms over taping is that they allow a compiler to optimise the generated reverse pass for the particular source program in question, as well as its composition with the forward pass, before they are run even once. Depending on the source language, an additional benefit could be somewhat lower memory usage (as the primal operations need not be stored, only their arguments). These algorithms are known as *define-then-run* reverse AD or *source-transform reverse AD*.

The primary downside of define-then-run reverse AD is that one needs to differentiate a much larger language than with taping AD: instead of only needing to arrange for a tape to be constructed and to compute partial derivatives for each primitive arithmetic operation, here one needs to be able to differentiate every individual language construct, including conditionals, loops, variable binding, etc. As a result, define-then-run reverse AD algorithms for different source languages can look rather different.

First-order languages. When differentiating first-order languages (i.e. languages without lambda abstraction, recall Section 2.1.1), define-then-run code transformations are not so different from the pattern set by the graph algorithm of Section 2.2.6: one generates code that computes the forward pass, which saves various intermediate results, and for the reverse pass, which uses those saved intermediate results. The most important differentiator between reverse AD algorithms for such languages is how structured their primal storage is.

One approach is to store the primals in a very structured way: intermediate values of the forward pass are stored as normal variables in the differentiated program that are in scope in the reverse pass, where they are used. To make this work, the forward pass for a conditional "exports" the to-be-saved primals out of both conditional branches; the reverse pass then contains a conditional again, whose branches pick out the appropriate saved primals to be used inside the reverse pass code of the branches. Sequential (while-)loops require creating an array of primals-to-be-saved. This approach is taken in e.g. TensorFlow [Abadi et al. 2016, §3.4, 4.1] and JAX.

To avoid complicated threading-through of primals, or to allow more interesting control flow, a different approach is to store primals in an *unstructured* way, despite doing define-then-run reverse AD. Tapenade [Hascoët and Pascual 2013] is a code transformation on Fortran and C in this style: forward and reverse pass code is generated directly from the source program, but primals are stored on a single stack that exists for the duration of one reverse derivative evaluation. In the forward pass, primals are pushed on this stack; when the corresponding reverse pass code is run and the primals are required, they are popped from the stack. To reduce the runtime overhead of this dynamic stack, Tapenade also includes various optimisations (checkpointing, as well as various analyses for reducing the number of tape stores [Hascoët et al. 2005]).

Higher-order languages. Handling higher-order languages means handling lambda abstraction and application. If user-defined functions are always closed (i.e. have no free variables), backpropagating through a call to such a function (i.e. visiting a function call operation during the reverse pass) only produces cotangent contributions to the function argument, which is a term readily available at the call site in question. Therefore, the data flow is fully clear, at least locally, and the only algorithm design challenge is ensuring that one can do a reverse pass through the function body at each call site.

If user-defined functions may be open, they can use also values from the lexical context of the lambda abstraction (i.e. the values in its closure). This lexical context is unavailable at the call site, so the cotangent contributions to the closure must be *kept* until the reverse pass reaches the lambda abstraction that created this function, at which point the closure cotangent contributions can be backpropagated to the correct computation nodes. The difficulty here is that the makeup of this closure is not evident from the type of the function. For example, consider the following program to be differentiated:

$$
\begin{aligned}
f\ x = \mathbf{let}\ & g = \lambda y.\ y + 1 && \text{①}\\
& h = \mathbf{let}\ z = \text{(some expensive computation)}\\
& \quad\ \mathbf{in}\ \lambda y.\ z \cdot y && \text{②}\\
\mathbf{in}\ & g\ (2 \cdot x) + h\ (x + 1)
\end{aligned}
$$

Both g and h have type $\mathbb{R} \to \mathbb{R}$ (as does f), but g is closed whereas h references z in its closure — a variable that is not in scope any more at the call site of h. Thus, while nothing needs to be backpropagated to the lambda at ①, the call to h on the last line should produce a cotangent to z and backpropagate it to the lambda at ②. In general, the cotangent backpropagated to a value of type $\sigma \to \tau$ thus necessarily has existential type (assuming a typed (functional) language); this is not unlike how existential types appear when doing typed closure conversion [Minamide et al. 1996, §2].

These ideas are implemented in an untyped language (Scheme) by Pearlmutter and Siskind [2008] and applied to a typed language by Vytiniotis et al. [2019]. The CHAD algorithm, described in detail in Chapter 5, is similar to the presentation by Vytiniotis et al. but has a correctness proof based on category theory [Vákár and Smeding 2022]; afterwards, Nunes and Vákár [2023] extended this formal analysis to more expressive functional languages. We analyse the algorithm from an operational point of view in Chapters 6 and 7.

Another notable reverse AD implementation in the define-then-run style is Enzyme [Moses and Churavy 2020], which differentiates the intermediate representation (IR) of the LLVM[35] compiler pipeline (LLVM IR). Enzyme does not need to handle open functions because functions are always closed on the level of LLVM IR — any closures that existed in the higher-level language have already been compiled away using e.g. closure conversion.

On Futhark, an array language with second-order array combinators but no unrestricted lambda abstraction, Schenck et al. [2022] provide a reverse AD algorithm that is primarily designed to produce a differentiated program with data flow and program structure that is as simple as possible, for the benefit of compiler optimisations running afterwards. In pursuit of this goal, they introduce a relatively significant amount of primal recomputation in the reverse pass. As a result, their time complexity is not optimal for reverse AD (recall 'Complexity of reverse AD' on page 32), although the amount of recomputation is bounded by a property of the source program that is relatively low in practice.

Optimisations. Because define-then-run reverse AD algorithms are less similar to each other than taping algorithms, it is harder to formulate optimisations that apply to many algorithms. Checkpointing, however, is general enough that it still applies; optimality proofs for checkpointing schemes apply only to straight-line programs [Grimm et al. 1996], but more general schemes exist [Siskind and Pearlmutter 2018].

Define-then-run versus define-by-run. Firstly, while most AD algorithms can be categorised as mostly one of the two upon inspection ("How much structure of the target program is exposed to the compiler?"), the terms 'define-then-run' and 'define-by-run' have no rigorous definitions. More interestingly, however, we are aware of very little work specifically comparing these two styles of doing AD for their practical strengths and weaknesses. One (very recent) article indirectly discussing this is [Huang et al. 2026], which considers a first-order language with arrays and unrestricted mutation, and observes significant performance improvement from using a structured tape (statically separated into fragments

[35] https://llvm.org/

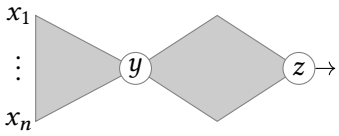
used by the various parts of the program) instead of an unstructured, linear tape as in taping reverse AD. The reason they cite is amenability for compiler optimisations after AD, which is precisely our motivation for pursuing define-then-run approaches in the second half of this thesis.

While we do study define-then-run (Chapters 5 to 7) and define-by-run (Chapter 3 and, to an extent, Chapter 4) algorithms in this thesis, we are not in a position to give a conclusive answer to the comparison question either: we have no comprehensive and compatible benchmarks between the algorithms, and in any case the only real define-by-run algorithm we study (dual-numbers reverse AD in Chapter 3) has performance problems unrelated to the overall style.

2.2.11 Other topics in automatic differentiation

While this thesis focuses on a rather traditional problem statement (reverse AD for first-order derivatives, although applied to challenging source languages), various generalisations and extensions to AD have been formulated in the literature. We briefly discuss some important ones.

Mixed-mode AD. Suppose one wishes to compute the full Jacobian of a program with a data-flow graph that looks as follows (with n large):



(2.18)

where the shaded areas stand for large amounts of nodes in roughly the indicated connection structure. The values x_i, y, z are scalars. As this graph has n inputs and only one output, reverse AD is certainly the appropriate choice if the only other option is forward AD. However, because forward AD is typically significantly faster than reverse AD if the number of inputs and outputs is equal, a better choice here may be a more complicated scheme: compute $\frac{\partial z}{\partial y}$ using forward AD, compute the gradient $\left(\frac{\partial y}{\partial x_1}, \ldots, \frac{\partial y}{\partial x_n}\right)$ using reverse AD, and finish with a scalar–vector product of the two to obtain $\left(\frac{\partial z}{\partial x_1}, \ldots, \frac{\partial z}{\partial x_n}\right)$. This has the downside over plain reverse AD of requiring n additional multiplications, but the potential significant upside of differentiating a large part of the function in forward mode, with the attendant lower memory requirements and lower administration overhead.

A common, if somewhat oversimplified,[36] way of illustrating the essence of

[36]While this perspective is elegant and concise, real AD algorithms do not treat the input program as a long composition of operations on a large state vector; for this reason, the author's opinion is that this perspective is not very helpful for understanding the design of AD algorithms.

AD starts by expressing a function as a composition of small steps:

$$f = f_n \circ f_{n-1} \circ \ldots \circ f_2 \circ f_1$$

Then one observes that computing and then multiplying together all the individual Jacobian matrices is very slow, especially when dense but even when using sparse matrices (primal arguments of the Jacobians are elided here for conciseness):

$$Jf = Jf_n \cdot Jf_{n-1} \cdot \ldots \cdot Jf_2 \cdot Jf_1$$

One then describes forward and reverse AD as adding a vector on one side of the product and positioning the parentheses such that all multiplications are matrix–vector multiplications (we may do this because matrix multiplication is associative):

$$\begin{aligned} Df\ \vec{x}\ \vec{u} &= Jf_n \cdot (Jf_{n-1} \cdot (\ldots \cdot (Jf_2 \cdot (Jf_1 \cdot \vec{u})))) \\ (Df)^\top\ \vec{x}\ \vec{u} &= ((((\vec{u}^\top \cdot Jf_n) \cdot Jf_{n-1}) \cdot \ldots) \cdot Jf_2) \cdot Jf_1 \end{aligned}$$

In this context, mixed-mode AD refers to some placement of the parentheses that is neither of these two.

First-class support for mixed-mode AD is rare in implementations, as automatic determination of the optimal complex strategy is NP-complete [Naumann 2008] and because implementations do generally allow a user to give a custom, manually specified derivative for parts of their program; with this functionality, a user can achieve most of mixed-mode AD manually by just specifying a custom derivative computed using a different, standard AD mode. For example, the complex strategy described above for Eq. (2.18) could be achieved by differentiating the program using reverse AD and specifying a custom derivative for the second half, itself computed using forward AD.

Higher-order derivatives. There are various use cases where f' is not enough, and one needs second-order (f'') or higher-order derivatives (f''', etc.). Examples are optimisation algorithms like Newton optimisation and Laplace approximation in probabilistic programming.

There are multiple approaches to computing such higher-order derivatives using AD. One option is to compute more coefficients of the Taylor series of the function: the dual-numbers forward AD algorithm of Section 2.2.3 computed pairs $(f\ x, \frac{\partial f}{\partial x}(x))$, which one can see as the first two coefficients of the Taylor series of f at x; the Faà di Bruno formula, a generalisation of the chain rule to higher-order derivatives, shows how to compute higher-order partial derivatives in a similar, still fully compositional fashion. [Huot et al. 2022]

Usually, however, one needs a specific operation on higher-order derivatives that is more efficiently served by combining standard AD modes. For example,

Newton optimisation requires not a full Hessian matrix, but only the ability to compute a Hessian–vector product. This can be computed using reverse-over-forward or forward-over-reverse equally, given $f : \mathbb{R}^n \to \mathbb{R}$:

$$(D(\vec{x} \mapsto Df\ \vec{x}\ \vec{u}))^\top\ \vec{x}\ 1 = Hf\ \vec{x} \cdot \vec{u} \approx (Hf\ \vec{x})^\top \cdot \vec{u} = D(\vec{x} \mapsto (Df)^\top\ \vec{x}\ 1)\ \vec{x}\ \vec{u}$$

where $Hf : \mathbb{R}^n \to \mathbb{R}^{n \times n}$ is the Hessian matrix of f at a particular input point, and the '$\approx$' stands for equality if $Hf\ \vec{x}$ is symmetric, which is true if f is differentiable at $\vec{x}$. Which of the two strategies (reverse-over-forward or forward-over-reverse) is more appropriate depends on the AD implementations and the function f. For details and references, see [Margossian 2019, §5.3].

Sparse differentiation. Suppose that we want to compute the full Jacobian matrix of the following function, given some closed $f : \mathbb{R} \to \mathbb{R}$:

$$\begin{aligned} &g : \mathbb{R}^n \to \mathbb{R}^n \\ &g\ \vec{x} = \text{map}\ f\ \vec{x} \end{aligned}$$

Naively, both forward AD and reverse AD would require n passes, as both the codomain and the domain of g contain n scalars. However, the Jacobian in question, Jg, is actually very sparse: (assume f is defined $f\ u = \ldots$)

$$Jg\ \vec{x} = \begin{pmatrix} \frac{\partial f}{\partial u}(x_1) & & \mathbf{0} \\ & \ddots & \\ \mathbf{0} & & \frac{\partial f}{\partial u}(x_n) \end{pmatrix}$$

If we compute $Jg\ \vec{x}$ the naive way by applying g's forward derivative n times, we wastefully compute a lot of zeros (and using the reverse derivative instead does not help whatsoever — we would just compute rows instead of columns):

$$Jg\ \vec{x} = \begin{pmatrix} | & & | \\ (Df\ \vec{x}\ (1, 0, \ldots, 0)) & \cdots & (Df\ \vec{x}\ (0, \ldots, 0, 1)) \\ | & & | \end{pmatrix}$$

It is much more efficient to exploit the fact that the forward derivative (and hence forward AD) computes a *linear combination* of the columns of the Jacobian:

$$\begin{aligned} Df\ \vec{x}\ (1, \ldots, 1) &= Df\ \vec{x}\ (1, 0, \ldots, 0) + \cdots + Df\ \vec{x}\ (0, \ldots, 0, 1) \\ &= \left(\tfrac{\partial f}{\partial u}(x_1), \ldots, \tfrac{\partial f}{\partial u}(x_n)\right) \end{aligned}$$

From these values, one can directly reconstruct the full Jacobian. This works because each row in $Jg\ \vec{x}$ happens to have only one non-zero value, so computing

the sum of the columns still computes all interesting values. In this case, one could do the same using reverse AD because also each *column* only has one non-zero value, but for the same number of invocations (one, in this case), forward AD tends to be faster in practice than reverse AD.[37]

More generally, one can exploit the sparsity pattern of a Jacobian to compute it, or parts of it, using fewer applications of forward or reverse AD than would naively be required. Methods for deducing sparsity patterns and deriving efficient AD invocations from them are sometimes called *automatic sparse differentiation*, or simply sparse Jacobian computation. For background and references, we refer to [Hill et al. 2025].

[37]Note that this discussion is separate from how to differentiate through 'map' *inside* forward or reverse AD, as we did in Section 2.2.7. There, we were given an input and a cotangent and had to propagate it through the 'map', while here we cleverly choose the (co)tangent(s) to start with.

3 Dual-Numbers Reverse AD

As we have seen in Section 2.2.3, dual-numbers forward AD is a simple and effective way to compute forward derivatives that extends easily to a variety of language features. It owes this extensibility to the fact that it is mostly agnostic of the programming language: it just provides an alternative implementation of scalar arithmetic that additionally tracks forward derivatives. The technique is also easy to analyse semantically, with a correctness proof based on logical relations both on paper [Huot et al. 2020; Lucatelli Nunes and Vákár 2024] and a formalised one in the interactive theorem prover Rocq [Chin Jen Sem 2020]. Finally, dual-numbers forward AD is even easy to get practically efficient: in an array language, a (standard) struct-of-arrays transform would be desirable, as well as some known loop optimisation techniques as explained by Shaikhha et al. [2019], but few special tricks are required.

It is tempting to try to get all these beautiful characteristics for reverse AD too. Huot et al. [2020] present a continuation-based reverse-AD-like algorithm as an application of their forward AD proof method, but with the knowledge that its complexity is quite bad — it computes the correct result semantically, but has exponential recomputation behaviour. Brunel et al. [2020] do better and present a proven complexity-efficient reverse-AD analogue of dual-numbers AD on a simply-typed lambda calculus; their approach is at heart the same as that of Huot et al. [2020], but they fix the complexity by simplifying call trees at runtime using algebraic linearity: if f is linear, then $f\ x + f\ y = f\ (x + y)$, saving one call to f. They call this simplification *linear factoring*. However, they accomplish this by using a target language with a custom operational semantics, and it is unclear whether that target language itself can be implemented efficiently on

This chapter is based on [Smeding and Vákár 2025] (published in JFP), which is an extended and revised version of the previous [Smeding and Vákár 2023a], published at POPL 2023. Compared to the journal paper, this chapter has a rewritten introduction, somewhat abridged *Key ideas* (3.1), minor textual updates elsewhere, and updated notation for consistency with the rest of the thesis.

actual hardware.

In this chapter, we present a complexity-efficient implementation of the algorithm of Huot et al. [2020] (and Brunel et al. [2020]) by optimising it, step-by-step, to an a-priori completely different AD approach: taping. The central AD-specific optimisation that we apply is based on the linear factoring idea of Brunel et al. [2020]; additionally, we use Cayley transformation (a generally applicable functional programming technique also known as "difference lists", explained and discussed in Section 3.5), simple sparse vectors, and functional in-place updates to lose log-factors in the complexity.

Taping has been applied in a functional context in the Haskell library 'ad' [Kmett and contributors 2021], and Krawiec et al. [2022] explain why this library works using a step-by-step derivation similar to the one in this chapter. While this chapter has similar start and end points to the work of Krawiec et al. [2022], we have different steps and a different presentation that focuses on the complexity rather than semantical correctness. A comparison can be found in Section 3.13.1.

In addition to this novel link between existing algorithms, we also add support for task parallelism: we differentiate user-annotated fork-join parallelism in the source program to parallelism in the corresponding parts of the forward and reverse pass of the derivative program. This support for parallelism is conceptually straightforward and could apply similarly to most taping-based AD algorithms.

Summary of contributions. Concretely, our main contributions are as follows:

- We show how the theoretical analysis of Brunel et al. [2020] 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 3.4.

- We show how this algorithm can be made complexity-efficient by using the standard functional programming techniques of Cayley transformation (Section 3.5) and (e.g. linearly typed or monadic) functional in-place updates (Section 3.7).

- We explain how our algorithm relates to classical approaches based on taping (Section 3.8).

- We demonstrate that, in contrast with previous similar approaches [Kmett and contributors 2021; Krawiec et al. 2022; Smeding and Vákár 2023a], 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 consume it in the dual pass to produce a task-parallel derivative (Section 3.10).

$$\lambda\langle x : \mathbb{R}, y : \mathbb{R}\rangle.$$
$$\quad \textbf{let } z = x + y$$
$$\quad \textbf{in } x \cdot z$$

(a) Original

$$\lambda\langle\langle x : \mathbb{R}, dx : \underline{\mathbb{R}}\rangle, \langle y : \mathbb{R}, dy : \underline{\mathbb{R}}\rangle\rangle.$$
$$\quad \textbf{let } \langle z, dz\rangle = \langle x + y, dx + dy\rangle$$
$$\quad \textbf{in } \langle x \cdot z, x \cdot dz + z \cdot dx\rangle$$

(b) Dual-numbers forward AD

$$\lambda\langle\langle x : \mathbb{R}, dx : \underline{\mathbb{R}} \multimap \underline{\mathbb{R}} \times \underline{\mathbb{R}}\rangle, \langle y : \mathbb{R}, dy : \underline{\mathbb{R}} \multimap \underline{\mathbb{R}} \times \underline{\mathbb{R}}\rangle\rangle.$$
$$\quad \textbf{let } \langle z, dz\rangle = \langle x + y, \underline{\lambda}(d : \underline{\mathbb{R}}).\ dx\ d + dy\ d\rangle$$
$$\quad \textbf{in } \langle x \cdot z, \underline{\lambda}(d : \underline{\mathbb{R}}).\ dz\ (x \cdot d) + dx\ (z \cdot d)\rangle$$

(c) Dual-numbers reverse AD

Figure 3.1: 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 $\mathbb{R} \times \mathbb{R} \to \mathbb{R}$.

- We give an implementation of the parallelism-ready algorithm of Section 3.10 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 3.11).
- 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 3.13). We also briefly describe the broader relationship with related work.

3.1 Key ideas

Naive dual-numbers reverse AD. The design idea of the naive (extremely inefficient) reverse AD version of dual-numbers forward AD is to replace the tangent scalar, paired together with every source-program scalar, with a backpropagator.[1] Let us look at an example. The small program in Fig. 3.1a has been differentiated by a standard dual-numbers forward AD transform to Fig. 3.1b; this term takes tangents to the inputs (of type $\underline{\mathbb{R}}$) and returns corresponding tangents to the outputs (only one, in this case). The form of this program should be as expected for readers of Chapter 2 (in particular Section 2.2.3).

The reverse AD variant is shown in Fig. 3.1c; the tangent scalars (dx and dy) have been replaced with *backpropagators*. A backpropagator for a scalar x takes a cotangent to x and returns the corresponding cotangents to the input

[1] In this chapter, and all subsequent ones, the word *scalar* refers to a real value, theoretically continuous but in practice represented using a floating-point value.

$$
\begin{array}{l}
\lambda(x_0 : \mathbb{R}).\\
\quad \textbf{let } x_1 = x_0 + x_0\\
\quad \textbf{in let } x_2 = x_1 + x_1\\
\quad\quad \vdots\\
\quad \textbf{in let } x_{n+1} = x_n + x_n\\
\quad \textbf{in } x_{n+1}
\end{array}
\quad\rightsquigarrow\quad
\begin{array}{l}
\lambda\langle x_0 : \mathbb{R}, dx_0 : \underline{\mathbb{R}} \multimap \underline{\mathbb{R}}\rangle.\\
\quad \textbf{let } \langle x_1, dx_1\rangle =\\
\quad\quad\quad \langle x_0 + x_0, \underline{\lambda}(d : \underline{\mathbb{R}}).\ dx_0\ d + dx_0\ d\rangle\\
\quad \textbf{in let } \langle x_2, dx_2\rangle =\\
\quad\quad\quad \langle x_1 + x_1, \underline{\lambda}(d : \underline{\mathbb{R}}).\ dx_1\ d + dx_1\ d\rangle\\
\quad\quad \vdots\\
\quad \textbf{in let } \langle x_{n+1}, dx_{n+1}\rangle =\\
\quad\quad\quad \langle x_n + x_n, \underline{\lambda}(d : \underline{\mathbb{R}}).\ dx_n\ d + dx_n\ d\rangle\\
\quad \textbf{in } \langle x_{n+1}, dx_{n+1}\rangle
\end{array}
\qquad
\begin{array}{c}
dx_0\\
\nwarrow\ \nwarrow\\
dx_1\\
\nwarrow\ \nwarrow\\
\vdots\\
\nwarrow\ \nwarrow\\
dx_n\\
\nwarrow\ \nwarrow\\
dx_{n+1}
\end{array}
$$

Figure 3.2: 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.

(i.e. the gradient). More precisely, if $f : \sigma \to \mathbb{R}$ is the implicit function that computes x given the input of the program (of type σ), then the backpropagator dx for x has type[2] $dx : \underline{\mathbb{R}} \multimap \underline{\sigma}$ and satisfies $dx\ u = (Df)^\top\ x\ u$, using notation from Section 2.2.2. In particular, if the full original program has type $\sigma \to \mathbb{R}$, then the dual-numbers reverse-AD-transformed program will return a single backpropagator that, when passed '1', computes the gradient of the program at the given input.

Note that the expected reversal of reverse AD has been achieved: calling the backpropagator for the result of Fig. 3.1c yields calls to dz and dx, and dz itself then proceeds to call the input backpropagators.

The code transformation that produces Fig. 3.1c from Fig. 3.1a, shown in full in Fig. 3.6, is simple and it is easy to see that it is correct via a logical relations argument [Lucatelli Nunes and Vákár 2024; Huot et al. 2020]. The superscript '1' to **D** indicates the version of the transformation (later improved versions have higher indices); the subscript 'c' is the type of the input to the top-level program (i.e., $\mathbb{R} \times \mathbb{R}$ in Fig. 3.1). The transformation assumes that the top-level input and output of the program contain only scalars, discrete types and products; generalisation to more zeroth-order types (coproducts and recursive, parametrised data types, i.e. parametrised algebraic data types) are given in Section 3.9.[3]

Efficiency. The dual-numbers reverse AD algorithm described here inherits some useful properties of dual-numbers forward AD; in particular, we retain

[2] We use '$\multimap$' to emphasise that the backpropagator is algebraically linear. Operationally, $\multimap = \to$.

[3] We study a version of this algorithm that efficiently supports arrays in Chapter 4.

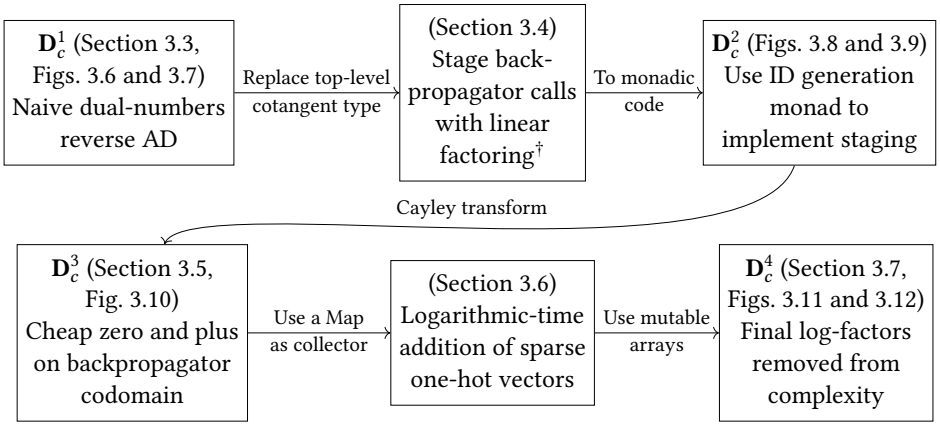


Figure 3.3: Overview of the optimisations to dual-numbers reverse AD as a code transformation that are described in this chapter. († = inspired by [Brunel et al. 2020])

its generalisability (due to being mostly agnostic of the programming language apart from scalar operations) and its simplicity for analysis with formal proof (as discussed above). 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 Fig. 3.2 takes time *exponential* in n. Such overhead would make reverse AD completely useless in practice — particularly because other (less flexible) reverse AD algorithms exist that indeed do a lot better. (See e.g. [Griewank and Walther 2008; 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 chapter presents a sequence of changes to the code transformation (see the overview in Fig. 3.3) 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 Fig. 3.3 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 Fig. 3.3 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 Fig. 3.6, we see that in the end all we produce is a giant sum of applications of backpropagators to scalars; hence, in this giant sum, we should be able to contract applications 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 (the type of the program input[4]) from our backpropagators, but instead a c wrapped in an object (written 'Staged c') that can delay calls to linear functions producing a c. That is to say: the backpropagators now have type $\underline{\mathbb{R}} \multimap \text{Staged}\, c$, not $\underline{\mathbb{R}} \multimap c$. 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 the order in which we do so is important for efficiency: it turns out that we are able to maximally apply linear factoring only if we call the backpropagators in reverse dependency order.

We thus need an identity witness, a linear order and a witness for the dependency order. It turns out that for *sequential* input programs, it suffices to use the same order for all three: we generate a unique, strictly monotonically increasing identifier (ID) for each backpropagator that we create, giving identity, linear order ($<$) and chronological order of creation, which is stronger than the dependency order and thus sufficient for our purposes. We generate these IDs by letting the differentiated program run in an ID generation monad (a special case of a state monad). The result is shown in Fig. 3.8, which is very similar to the previous version in Fig. 3.6 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 and resolving[5] those calls in order from highest to lowest ID, we delay the calls as long as possible and we ensure that

[4] More precisely: cotangents to the program input. For zeroth-order types, these are equal.

[5] By unfolding one "layer" at a time: we delay subcalls too.

every backpropagator is called at most once. To see that this is true, consider 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), the relation between backpropagators given by $\{(f, g) \mid g \text{ directly mentions } f \text{ in its closure}\}$ defines a directed acyclic graph; this graph is the data flow graph (Section 2.2.4) of this run of the original program, and the linear order on the strictly monotonically increasing IDs defines a topological order on this graph. Thus, a backpropagator will only call other backpropagators with lower IDs, and resolving backpropagators from the highest to the lowest ID indeed calls every backpropagator at most once, fixing the most egregious complexity problem.

But we are not done yet. The code transformation at this point ($\mathbf{D}_c^2$ in Fig. 3.8) still has a glaring problem: orthogonal to the issue that backpropagators were called too many times (which we fixed), we are still creating very large input cotangents with many zeros and adding those together. This problem is somewhat more subtle, because it is not actually apparent in the program transformation itself; indeed, looking back at Fig. 3.1c, no such large cotangent values are apparent. However, the only way to *use* the program in Fig. 3.1c to do something useful, namely to compute the cotangent (gradient) of the input, is to pass $(\lambda z.\ \langle z, 0\rangle)$ to dx and $(\lambda z.\ \langle 0, z\rangle)$ to dy; it is easy to see that generalising this to larger input structures results in input values like $\langle 0, \ldots, 0, z, 0, \ldots, 0\rangle$ that get added together. Values of this form are called *one-hot tuples*; in the transformation, 'Wrap' in Fig. 3.7 is responsible for creating them. 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 3.5) and using a better sparse vector representation (Map $\mathbb{Z}\ \underline{\mathbb{R}}$ instead of a plain c value; Section 3.6). 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 chapter) 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 3.7). 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 3.7.1). (The latter can be generalised to the parallel case; see below.)

At this point we are done for sequential programs, because we have 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 [Lucatelli Nunes and Vákár 2024];

2. Our optimisations are semantics-preserving: the custom linear factoring optimisation preserves semantics because derivatives (backpropagators) are linear functions, and the remaining steps are standard optimisations already known to preserve semantics (namely sparsity via Cayley transformation[6], a tree map for additional explicit sparsity, and usage of a mutable array to optimise that tree map).

Parallelism. If the user is satisfied with a fully sequential (non-parallel) computation of the derivative, the strictly monotonically increasing integers (with their standard linear order) used so far suffice as backpropagator IDs.

However, if the source program has (task) parallelism (we assume fork-join parallelism in this chapter), 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 thesis.[7]

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

[7] It is also treacherous ground: it turns out to be very difficult to fulfil the promise of functional

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.[8] We assume parallelism in the source program is expressed using a parallel pair constructor with the following typing rule:

$$\frac{\Gamma \vdash t : \tau \qquad \Gamma \vdash s : \sigma}{\Gamma \vdash t \star s : \tau \times \sigma}$$

(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. For details, see Section 3.10.

Relation to taping. Finally, we are now in the position to note the similarity to (sequential[9]) taping-based AD as in [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 3.7), which reifies this tape in a large array in the reverse pass, eliminates also this last difference, especially if one then proceeds to already use this array in the forward pass (Section 3.8.3).

3.2 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.

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].)

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

[9] 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.

Typing forward AD. For forward AD on first-order programs (or at least, programs whose input and output do not contain function values), the desired type seems quite evident: $\mathcal{F} : (\sigma \to \tau) \rightsquigarrow (\sigma \times \underline{\sigma} \to \tau \times \underline{\tau})$, 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 chapter; in an implementation one can take $\underline{\tau} = \tau$, but there is some freedom in this choice.[10] Given a program $f : \sigma \to \tau$, $\mathcal{F}[f]$ is a program that takes, in addition to f's original argument, also a tangent at that point; the output is then f's normal result paired with the 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 $\mathbf{x} = (x_1, \ldots, x_n)$):[11]

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

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, $\mathrm{snd}(\mathcal{F}[f](x, dx))$ 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:

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

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

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

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

[10] For example, $\underline{\mathbb{R}} = \mathbb{R}$, but for $\underline{\mathbb{Z}}$ one can choose the unit type $\mathbf{1}$ and be perfectly sound and consistent.

[11] This generalises to more complex (but still zeroth-order (see Section 2.1.1)) in/outputs by regarding those as collections of real values as well.

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 $\sigma \times \underline{\sigma} \to \tau \times \underline{\tau}$ should really[12] 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 $\mathcal{R}_1$. To be compatible with (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 $\mathcal{R}_1 : (\Gamma \vdash t : \tau) \rightsquigarrow (\Gamma, d : \underline{\tau} \vdash \mathcal{R}_1[t] : \tau \times \underline{\Gamma})$, where $\underline{\Gamma}$ is a tuple containing the derivatives of all elements in the environment Γ. (To be precise, we define $\underline{\varepsilon} = \mathbf{1}$ for the empty environment and $\underline{\Gamma, x : \tau} = \underline{\Gamma} \times \underline{\tau}$ 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:

$$\Gamma, d : \underline{\tau} \vdash \mathcal{R}_1[\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2] : \tau \times \underline{\Gamma} \tag{3.1}$$

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

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

$$\Gamma, d : \underline{\sigma} \vdash \mathcal{R}_1[e_1] : \sigma \times \underline{\Gamma}$$
$$\Gamma, x : \sigma, d : \underline{\tau} \vdash \mathcal{R}_1[e_2] : \tau \times (\underline{\Gamma} \times \underline{\sigma})$$

To produce the program in (3.1), 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 $\mathcal{R}_2$:[13]

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

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:

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

[13] 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.

which is the type of the reverse[14] AD code transformation derived by Elliott [2018] and in CHAD (e.g. [Vákár and Smeding 2022]; see also Section 3.13.2).[15]

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 $\mathcal{F}$.

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 $\mathcal{R}_3$:

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

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 chapter; that is: it has a zero $\underline{0}$ as well as

[14]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 $\mathcal{F}$.

[15]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 zeroth-order in- and output types, the two coincide.

a commutative, associative addition operation $(+) : c \times c \to c$ 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 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 $\mathcal{R}_3[t]$:

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

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:

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

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 chapter.

Moving the pair to the leaves. Let us return to forward AD for a moment. Recall the type we gave for forward AD:[16]

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

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

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

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:

$$\mathcal{F}_{\text{dual}} : (\sigma \to \tau) \rightsquigarrow (\text{Dual}[\sigma] \to \text{Dual}[\tau])$$
$$\text{Dual}[\mathbb{R}] = \mathbb{R} \times \underline{\mathbb{R}} \qquad \text{Dual}[\mathbf{1}] = \mathbf{1} \qquad \text{Dual}[\sigma \times \tau] = \text{Dual}[\sigma] \times \text{Dual}[\tau]$$

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

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 $\mathbf{1}$ or $\mathbb{Z}$ alone. For the given example program f, dual-numbers forward AD would yield the following derivative program type:

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

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 $f : (\mathbb{R}_1 \times \mathbb{R}_2) \times \mathbb{R}_3 \to \mathbb{R}_4$)

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

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

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 $g : (\underline{\mathbb{R}_1} \times \underline{\mathbb{R}_2}) \times \underline{\mathbb{R}_3} \multimap c$ is linear, and hence e.g. $\lambda(x : \underline{\mathbb{R}_2}).\ g\ \langle\langle 0, x\rangle, 0\rangle$ 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 $t = \lambda(x : \sigma \times \tau).\ \text{fst}\ x$ of type $\sigma \times \tau \to \sigma$ and the types of its derivative using both methods:

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

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

$$\text{Dual}_c[\![\mathbb{R}]\!] = \mathbb{R} \times \underline{\mathbb{R}} \multimap c \qquad \text{Dual}_c[\![\mathbf{1}]\!] = \mathbf{1}$$

$$\text{Dual}_c[\![\sigma \times \tau]\!] = \text{Dual}_c[\![\sigma]\!] \times \text{Dual}_c[\![\tau]\!]$$

Their implementations look as follows:

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

$$\mathcal{R}_{3\text{dual}}[\![t]\!] = \lambda(x : \langle \text{Dual}_c[\![\sigma]\!], \text{Dual}_c[\![\tau]\!]\rangle).\ \text{fst}\ x$$

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 chapter in Section 3.5. However, in this chapter 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 3.9 and 3.10).

3.3 Naive, unoptimised dual-numbers reverse AD

We first describe the naive implementation of dual-numbers reverse AD: this algorithm is easy to define and prove correct compositionally, but it is wildly inefficient in terms of complexity. Indeed, it tends to blow up to exponential overhead over the original function, whereas the desired complexity is to have only a constant factor overhead over the original function. 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.

3.3.1 Source and target languages

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

The source language is defined in Fig. 3.4; the initial target language is given in Fig. 3.5. 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

Types:

$$\sigma, \tau ::= \mathbb{R} \mid \mathbf{1} \mid \sigma \times \tau \mid \sigma \to \tau \mid \mathbb{Z}$$

Terms:

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

Figure 3.4: The source language of all variants of this chapter's reverse AD transformation. $\mathbb{Z}$, the type of integers, is added as an example of a type that AD does not act upon.

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 Fig. 3.5, 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 a function value (in the input or output of a program) should be, we disallow function types on either side of the $\multimap$-arrow.[17] (Note that higher-order functions *within* the program are fine; the full program should just have zeroth-order input and output types.) Operationally, however, linear functions are just regular functions: the operational meaning of all code in this chapter remains identical if all $\multimap$-arrows are replaced with $\to$ (and partial derivative operations are allowed in regular terms).

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

Note that zero and plus will always be of a type that is (part of) the domain or codomain of a linear function, which therefore has the required commutative

[17] In Section 3.5 we will, actually, put endomorphisms ($a \to a$) on both sides of a $\multimap$-arrow; for justification, see there.

Types:

$$
\begin{array}{rcll}
\overline{\sigma}, \overline{\tau} & ::= & \mathbb{R} \mid \mathbf{1} \mid \overline{\sigma} \times \overline{\tau} \mid \mathbb{Z} & \text{(types without functions)} \\
\sigma, \tau & ::= & \mathbb{R} \mid \mathbf{1} \mid \sigma \times \tau \mid \mathbb{Z} \mid \sigma \to \tau & \\
& \mid & \overline{\sigma} \multimap \overline{\tau} & \text{(linear functions)}
\end{array}
$$

Terms:

$$
\begin{array}{rcll}
s, t & ::= & x \mid \langle\rangle \mid \langle s, t \rangle \mid \mathrm{fst}\ t \mid \mathrm{snd}\ t \mid \lambda(x : \tau).\ t \mid s\ t \mid \mathbf{let}\ x : \tau = s\ \mathbf{in}\ t & \\
& \mid & r \mid op(t_1, \ldots, t_n) & \\
& \mid & \underline{\lambda}(z : \overline{\tau}).\ b & \text{(linear lambda abstraction)}
\end{array}
$$

Linear function bodies:

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

Figure 3.5: The target language of the unoptimised variant of the reverse AD transformation. Components that are also in the source language (Fig. 3.4) are set in grey.

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 $\partial_i op(x)(d) = d \cdot \frac{\partial(op(x))}{\partial x_i}$.

3.3.2 The code transformation

The naive dual-numbers reverse AD algorithm acts homomorphically over all program constructs in the input program, except for those constructs that non-trivially manipulate real scalars. The full program transformation is given in Fig. 3.6. We use some syntactic sugar: **let** $\langle x_1, x_2 \rangle = s$ **in** t should be read as **let** $y = s$ **in let** $x_1 = \mathrm{fst}\ y$ **in let** $x_2 = \mathrm{snd}\ y$ **in** t, where y is fresh.

The transformation consists of a mapping $\mathbf{D}^1_c[\tau]$ on types τ and a mapping $\mathbf{D}^1_c[t]$ on terms t.[18] 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 $\mathbf{D}^1_c$-transformed program at some point computes a scalar–backpropagator pair $\langle x, d \rangle$ 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

[18] 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.

On types:

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

On environments:

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

On terms:

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

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

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

$\mathbf{D}_c^1[\langle\rangle] = \langle\rangle$ $\qquad$ $\mathbf{D}_c^1[\mathrm{fst}\ t] = \mathrm{fst}\ \mathbf{D}_c^1[t]$

$\mathbf{D}_c^1[\langle s, t\rangle] = \langle \mathbf{D}_c^1[s], \mathbf{D}_c^1[t]\rangle$ $\qquad$ $\mathbf{D}_c^1[\mathrm{snd}\ t] = \mathrm{snd}\ \mathbf{D}_c^1[t]$

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

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

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

Figure 3.6: The naive code transformation from the source (Fig. 3.4) to the target (Fig. 3.5) language. The cases where $\mathbf{D}_c^1$ just maps homomorphically over the source language are set in grey.

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 $\partial_i op(x_1,\ldots,x_n)(z) = z \cdot \frac{\partial(op(x_1,\ldots,x_n))}{\partial x_i}$, the appropriate backpropagator to return is indeed $\underline{\lambda}(z : \underline{\mathbb{R}}).\ \sum_{i=1}^{n} d_i\ (\partial_i op(x_1,\ldots,x_n)(z))$ as is written in Fig. 3.6. 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 3.2:

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

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

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

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

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

and from there to $\mathcal{R}_2$. The conversion from $\sigma \times (\sigma \multimap c)$ to $\mathbf{D}^1_c[\sigma]$, for zeroth-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 Fig. 3.7. The final conversion from $\mathcal{R}_3$ to $\mathcal{R}_2$ is easy in the simply-typed world (as described in Section 3.2); this conversion is implemented in the top-level wrapper, Wrap[1], also in Fig. 3.7.

3.3.3 Running example

Let us look at the simple example from Fig. 3.1a in Section 3.1:

$$\lambda\langle x : \mathbb{R}, y : \mathbb{R}\rangle.\ \underbrace{\mathbf{let}\ z = x + y\ \mathbf{in}\ x \cdot z}_{t} \tag{3.2}$$

We have $x : \mathbb{R}, y : \mathbb{R} \vdash t : \mathbb{R}$. The code transformation $\mathbf{D}^1_c$ from Fig. 3.6 maps t to:

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

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

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

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

Figure 3.7: Wrapper around $\mathbf{D}^1_c$ of Fig. 3.6.

which satisfies $x : \mathbb{R} \times (\underline{\mathbb{R}} \multimap c), y : \mathbb{R} \times (\underline{\mathbb{R}} \multimap c) \vdash \mathbf{D}^1_c[t] : \mathbb{R} \times (\underline{\mathbb{R}} \multimap c)$. (We α-renamed z from Fig. 3.6 to z' here.) The wrapper Wrap^1 in Fig. 3.7 computes, given $x : \mathbb{R} \times \mathbb{R}$:

$$\begin{aligned}\mathrm{Interleave}^1_{\mathbb{R}\times\mathbb{R}}\ \langle x, \underline{\lambda}(z : \underline{\mathbb{R}} \times \underline{\mathbb{R}}).\ z\rangle = \langle &\langle \mathrm{fst}\ x, \underline{\lambda}(z : \underline{\mathbb{R}}).\ \langle z, \underline{0}\rangle\rangle \\ &, \langle \mathrm{snd}\ x, \underline{\lambda}(z : \underline{\mathbb{R}}).\ \langle \underline{0}, z\rangle\rangle\rangle\end{aligned}$$

The x and y in Eq. (3.2) 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 3.4.1 and 3.5.2, we will revisit this example to show how the outputs change.

3.3.4 Complexity of the naive transformation

Reverse AD transformations like the one described in this section are well-known to be correct [e.g. Brunel et al. 2020; Mazza and Pagani 2021; Huot et al. 2020; 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 $\mathrm{Wrap}^1[\lambda(x : \sigma).\ t : \tau] : \sigma \to \tau \times (\tau \multimap \sigma)$ 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. Fig. 3.2 in Section 3.1 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 a higher-order language like ours, 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,[19] 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 chapter, 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 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.

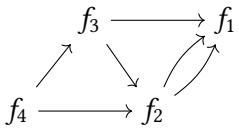
3.4 Linear factoring by staging function calls

As observed above in Section 3.3.4, the most important complexity problem of the reverse AD algorithm is solved if we ensure that all backpropagators are invoked at most once, and for that we must use that 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:

[19] Obstacles to this are e.g. $\underline{0}$ and $(+)$ on the type c; we will fix this in Sections 3.5 to 3.7.

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



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.

To obtain something like this evaluation order, the first thing that we must do 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 postponement, 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.[20] 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 that we replace what are now applications of linear functions with the creation of a dictionary containing one key-value (function-argument) pair, and that we 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 $\mathbf{D}^1_c[\mathbb{R}] = \mathbb{R} \times (\underline{\mathbb{R}} \multimap c)$ to instead read $\mathbf{D}^1_c[\mathbb{R}] = \mathbb{R} \times (\underline{\mathbb{R}} \multimap \text{Staged}\, c)$, where 'Staged c' is our "dictionary".[21] We might 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)

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

Suspending disbelief about implementability, this type can represent both literal c values (necessary for the one-hot vectors returned by the input backpropagators

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

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

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 $\langle x, \{f_1 \mapsto a_1, f_2 \mapsto a_2\}\rangle$ 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 $0_{\text{Staged}} \coloneqq \langle \underline{0}, \{\}\rangle : \text{Staged}\, c$.
- $(+) : c \to c \to c$ 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:

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

- 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 $\text{SCall}\ f\ x \coloneqq \langle \underline{0}, \{f \mapsto x\}\rangle : \text{Staged}\, c$. 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 resolution 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 it in the Map. For example, in the example at the beginning of this section, $f_4\ 1$ returns $\langle \underline{0}, \{f_2 \mapsto 1, f_3 \mapsto 2\}\rangle$. At this point, "resolving f_3" means calling f_3 at 2, observing the return value $\langle \underline{0}, \{f_2 \mapsto 8, f_1 \mapsto 10\}\rangle$, and adding it to the remainder (i.e. without the f_3 entry) of the previous Staged c object to get $\langle \underline{0}, \{f_2 \mapsto 9, f_1 \mapsto 10\}\rangle$.

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 addressed with the same fix.

Resolve order. The backpropagators that occur in the derivative program (as produced by $\mathbf{D}_c^1$ from Fig. 3.6) are not just arbitrary functions. Indeed, taking the target type c of the input backpropagators to be equal to the input type σ of the original program (of type $\sigma \to \tau$), as we do in Wrap^1 in Fig. 3.7, 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 $\text{Interleave}_\sigma^1$ (Fig. 3.7) 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 $op \in \text{Op}_n$ 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, 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 3.10, 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:

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

We call the second component of a Staged c value, which has type Map $\mathbb{Z}$ $((\underline{\mathbb{R}} \multimap$ Staged $c) \times \underline{\mathbb{R}})$, 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 Figs. 3.6 and 3.7, is shown in Figs. 3.8 and 3.9.

New program transformation. In Fig. 3.8, the term transformation now produces a term in the ID generation monad ($\mathbb{Z} \to - \times \mathbb{Z}$); therefore, all functions in the original program will also need to run in the same monad. This gives the

On types:

$$\mathbf{D}_c^2[\mathbb{R}] = \mathbb{R} \times (\mathbb{Z} \times (\underline{\mathbb{R}} \multimap \text{Staged}\, c)) \qquad \mathbf{D}_c^2[\mathbb{Z}] = \mathbb{Z} \qquad \mathbf{D}_c^2[\mathbf{1}] = \mathbf{1}$$
$$\mathbf{D}_c^2[\sigma \to \tau] = \mathbf{D}_c^2[\sigma] \to \mathbb{Z} \to \mathbf{D}_c^2[\tau] \times \mathbb{Z} \qquad \mathbf{D}_c^2[\sigma \times \tau] = \mathbf{D}_c^2[\sigma] \times \mathbf{D}_c^2[\tau]$$

On terms:

$$\text{If } \Gamma \vdash t : \tau \text{ then } \mathbf{D}_c^2[\Gamma] \vdash \mathbf{D}_c^2[t] : \mathbb{Z} \to \mathbf{D}_c^2[\tau] \times \mathbb{Z}$$
$$\mathbf{D}_c^2[x : \tau] = \lambda i.\ x : \mathbf{D}_c^2[\tau] \times i$$
$$\mathbf{D}_c^2[\langle s, t\rangle] = \lambda i.\ \mathbf{let}\ \langle x, i'\rangle = \mathbf{D}_c^2[s]\ i\ \mathbf{in\ let}\ \langle y, i''\rangle = \mathbf{D}_c^2[t]\ i'\ \mathbf{in}\ \langle\langle x, y\rangle, i''\rangle$$
$$\mathbf{D}_c^2[\mathbf{let}\ x : \tau = s\ \mathbf{in}\ t] = \lambda i.\ \mathbf{let}\ \langle x : \mathbf{D}_c^2[\tau], i'\rangle = \mathbf{D}_c^2[s]\ i\ \mathbf{in}\ \mathbf{D}_c^2[t]\ i'$$
$$\textit{etc.}$$
$$\mathbf{D}_c^2[r] = \lambda i.\ \langle\langle r, \langle i, \underline{\lambda}(z : \underline{\mathbb{R}}).\ 0_{\text{Staged}}\rangle\rangle, i + 1\rangle$$
$$\mathbf{D}_c^2[op(t_1, \ldots, t_n)] =$$
$$\lambda i.\ \mathbf{let}\ \langle\langle x_1, d_1\rangle, i_1\rangle = \mathbf{D}_c^2[t_1]\ i\ \mathbf{in}\ \ldots\ \mathbf{in\ let}\ \langle\langle x_n, d_n\rangle, i_n\rangle = \mathbf{D}_c^2[t_n]\ i_{n-1}$$
$$\mathbf{in}\ \langle\langle op(x_1, \ldots, x_n)$$
$$, \langle i_n, \underline{\lambda}(z : \underline{\mathbb{R}}).\ \text{SCall}\ d_1\ (\partial_1 op(x_1, \ldots, x_n)(z)) +_{\text{Staged}} \cdots$$
$$+_{\text{Staged}}\ \text{SCall}\ d_n\ (\partial_n op(x_1, \ldots, x_n)(z))\rangle\rangle$$
$$, i_n + 1\rangle$$

Staged interface:

$$\text{Staged}\, c = c \times \text{Map}\ \mathbb{Z}\ ((\underline{\mathbb{R}} \multimap \text{Staged}\, c) \times \underline{\mathbb{R}})$$
$$0_{\text{Staged}} : \text{Staged}\, c$$
$$0_{\text{Staged}} = \langle \underline{0}, \{\}\rangle$$
$$(+_{\text{Staged}}) : \text{Staged}\, c \to \text{Staged}\, c \to \text{Staged}\, c$$
$$\langle c, m\rangle +_{\text{Staged}} \langle c', m'\rangle = \langle c + c', m \cup m'\rangle \quad -- \textit{linear factoring}$$
$$\text{SCotan} : c \multimap \text{Staged}\, c$$
$$\text{SCotan}\ c = \langle c, \{\}\rangle$$
$$\text{SCall} : \mathbb{Z} \times (\underline{\mathbb{R}} \multimap \text{Staged}\, c) \to \underline{\mathbb{R}} \multimap \text{Staged}\, c$$
$$\text{SCall}\ \langle i, f\rangle\ x = \langle \underline{0}, \{i \mapsto \langle f, x\rangle\}\rangle$$

Figure 3.8: The monadically transformed code transformation (from Fig. 3.4 to Fig. 3.5 plus Staged operations), based on Fig. 3.6. Grey parts are unchanged or simply monadically lifted.

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

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

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

Figure 3.9: The wrapper corresponding to Fig. 3.8, based on Fig. 3.7. Grey parts are simply monadically lifted.

second change in the type transformation (aside from $\mathbf{D}^2_c[\mathbb{R}]$, which now tags backpropagators with an ID): $\mathbf{D}^2_c[\sigma \to \tau]$ now produces a monadic function type instead of a plain function type.

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

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

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 chapter.

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 Fig. 3.7, has now become $\text{SCotan} : \underline{\sigma} \multimap \text{Staged}\, \underline{\sigma}$, which injects a cotangent into a Staged c object. Interleave^2 "splits" this function up into individual $\underline{\mathbb{R}} \multimap \text{Staged}\, \underline{\sigma}$ 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.[22] This is done in an additional function, SResolve, which can be written as follows:

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

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

3.4.1 Running example

In Section 3.3.3, 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 Fig. 3.1a. With the updated transformation from Fig. 3.8, we now get (with lambda-application redexes already simplified for readability):

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

The result of $\text{Interleave}^2_{\mathbb{R}\times\mathbb{R}}\ \langle x, \text{SCotan}\rangle\ 0$, as called from Wrap^2 in Fig. 3.9, is:

$$\langle\langle\langle \text{fst}\ x, \langle 0, d_{\text{in},0}\rangle\rangle, \langle \text{snd}\ x, \langle 1, d_{\text{in},1}\rangle\rangle\rangle, 2\rangle$$

where we abbreviated the input backpropagators as $d_{\text{in},0} = \underline{\lambda}(z : \underline{\mathbb{R}}).\ \langle\langle z, \underline{0}\rangle, \{\}\rangle$ and $d_{\text{in},1} = \underline{\lambda}(z : \underline{\mathbb{R}}).\ \langle\langle \underline{0}, z\rangle, \{\}\rangle$. Now, assuming that the input x is, say, $\langle 12, 13\rangle$

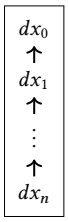
[22] Technically, some backpropagators (namely, the ones that appear in the top-level function output), are invoked more than once because Deinterleave^2 indiscriminately calls all output backpropagators. If the function output contains n scalars, this can lead to $O(n)$ overhead. (In particular, for 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.

and that the initial cotangent is 1, the Staged $(\mathbb{R} \times \mathbb{R})$ object that gets passed to SResolve in Wrap^2 (i.e. the result of $d\ z = d\ 1$ there) looks as follows:

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

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 $(+_{\text{Staged}})$.

3.4.2 Remaining complexity challenges



We have gained a lot with the function call staging so far: where the naive algorithm from Section 3.3 easily ran into exponential blowup of computation time if the results of primitive operations were used in multiple places, the updated algorithm from Figs. 3.8 and 3.9 completely solves this issue. For example, the program of Fig. 3.2 now results in the call graph displayed on the right: each backpropagator is called exactly once. However, some other complexity problems still remain.[23]

As discussed in Section 2.2.6 in the Background, 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 $\text{size}(x)$. Formally, 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 $\text{cost}(E)$ is the time taken to evaluate E, and $\text{size}(x)$ is the time taken to read all of x sequentially.

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

[23] Strictly speaking, this section does not prove that Wrap^2 does *not* have the correct complexity (although, in fact, it does not); rather, it argues that the expected complexity analysis does not go through. The same analysis *will* succeed for Wrap^4 after the improvements of Sections 3.5 to 3.7.

- Interleave^2 itself runs in $O(\text{size}(x))$. (The backpropagators it creates are more expensive, but those are not called just yet.)

- For the differentiated program, $\mathbf{D}^2_\sigma[P]$, we can see that in all cases of the transformation $\mathbf{D}^2_c$, the right-hand side does the work that P would have done, plus threading of the next ID to generate, as well as creation of backpropagators. Since this additional work is a constant amount per program construct, $\mathbf{D}^2_\sigma[P]$ runs in $O(\text{cost}(P\ x))$.

- Deinterleave^2 runs in $O(\text{size}(P\ x))$, i.e. the size of the program output; this is certainly in $O(\text{cost}(P\ x) + \text{size}(x))$ but likely much less.

Summarising, the primal pass as a whole runs in $O(\text{cost}(P\ x) + \text{size}(x))$, which is 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 Fig. 3.7, 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 Fig. 3.8) 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 Staged σ to call every scalar backpropagator (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[24] 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.

[24] 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.

- 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. $(\underline{\lambda}(z : \underline{\mathbb{R}}).\ \text{SCotan}\ \langle 0, \ldots, 0, z, 0, \ldots, 0\rangle)$ 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. $(\underline{\lambda}(z : \underline{\mathbb{R}}).\ \text{SCall}\ d_1\ (\partial_1 op(\ldots)(z)) +_{\text{Stg.}} \cdots +_{\text{Stg.}} \text{SCall}\ d_n\ (\partial_n op(\ldots)(z)))$, as created in $\mathbf{D}^2_\sigma[op(\ldots)]$. 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.)

Summarising again, 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 3.5) 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. This problem will become less pronounced after Cayley-transforming Staged c in Section 3.5: the backpropagators will then *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 3.6 by replacing the c in Staged c with a more efficient structure (namely, Map $\mathbb{Z}\ \underline{\mathbb{R}}$). This log-factor can optionally be further eliminated using mutable arrays as described in Section 3.7.

(C) The Map operations in SResolve are logarithmic in the size of the staging map. Like in the previous point, mutable arrays (Section 3.7) 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.

3.5 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 $(+_{\mathrm{Staged}})$). If we Cayley-transform Staged c, i.e. if we replace Staged c with Staged $c \to$ Staged c, all occurrences of 0_{Staged} in the code transformation turn into id and all occurrences of $(+_{\mathrm{Staged}})$ turn into $(\circ)$. Since id is a 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.

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);

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:[25] (the changed parts are shown in red)

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

The definition of $\text{Staged}\, c$ itself also gets changed accordingly:

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

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

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

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:

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

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 3.4, only changing $f\ a +_{\text{Staged}} \langle c, m'\rangle$ to $f\ a\ \langle c, m'\rangle$ on the last line: f from the Map now has type $\underline{\mathbb{R}} \multimap (\text{Staged}\, c \to \text{Staged}\, c)$.

3.5.1 Code transformation

We show the new code transformation in Fig. 3.10. 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.

[25] Despite the fact that we forbade it in Section 3.3.1, we are putting function types on both sides of a $\multimap$-arrow here. The monoid structure here is the one from the Cayley transform (i.e. with id and $(\circ)$). Notice that this monoid structure is indeed the one we want in this context: the "sum" (composition) of two values of type $(\text{Staged}\, c \to \text{Staged}\, c)$ corresponds with the sum (with $(+_{\text{Staged}})$) of the $\text{Staged}\, c$ values that they represent.

On types:

$$\mathbf{D}_c^3[\mathbb{R}] = \mathbb{R} \times (\mathbb{Z} \times (\underline{\mathbb{R}} \multimap (\text{Staged}\, c \to \text{Staged}\, c))) \qquad \mathbf{D}_c^3[\mathbb{Z}] = \mathbb{Z} \qquad \mathbf{D}_c^3[\mathbf{1}] = \mathbf{1}$$
$$\mathbf{D}_c^3[\sigma \to \tau] = \mathbf{D}_c^3[\sigma] \to \mathbb{Z} \to \mathbf{D}_c^3[\tau] \times \mathbb{Z} \qquad \mathbf{D}_c^3[\sigma \times \tau] = \mathbf{D}_c^3[\sigma] \times \mathbf{D}_c^3[\tau]$$

On terms:

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

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

Changed wrapper:

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

$\text{Deinterleave}_\tau^3 : \forall c.\ \mathbf{D}_c^3[\tau] \to \tau \times (\tau \multimap (\text{Staged}\, c \to \text{Staged}\, c))$
(Same as Deinterleave^2 in Fig. 3.9, except with id and $(\circ)$ in place of 0_{Staged} and $(+_{\text{Staged}})$)

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

Figure 3.10: The Cayley-transformed code transformation, based on Figs. 3.8 and 3.9. Grey parts are unchanged.

3.5.2 Running example

Apart from types, the only change in $\mathbf{D}_c^3$ since Section 3.4.1 is replacement of $(+_{\text{Staged}})$ by $(\circ)$. The result of $\text{Interleave}^3_{\mathbb{R}\times\mathbb{R}}\ \langle x, \text{SCotan}\rangle\ 0$ is the same as before, except that the input backpropagators now update a Staged $(\mathbb{R}\times\mathbb{R})$ pair instead of constructing one: $d_{\text{in},0} = \underline{\lambda}(z : \underline{\mathbb{R}}).\ \lambda\langle\langle dx, dy\rangle, m\rangle.\ \langle\langle z + dx, dy\rangle, m\rangle$ and $d_{\text{in},1} = \underline{\lambda}(z : \underline{\mathbb{R}}).\ \lambda\langle\langle dx, dy\rangle, m\rangle.\ \langle\langle dx, z + dy\rangle, m\rangle$. The Staged $(\mathbb{R}\times\mathbb{R})$ object passed to SResolve is now:

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

where f is a function that adds the key-value pairs $0 \mapsto \langle d_{\text{in},0}, z'\rangle$ and $1 \mapsto \langle d_{\text{in},1}, z'\rangle$ 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.

3.5.3 Remaining complexity challenges

In Section 3.4.2, 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 $\sigma = (\mathbb{R}\times\mathbb{Z})\times(\mathbb{R}\times\mathbb{R})$, the backpropagator for the second scalar looks as follows before and after the Cayley transform:

Before Cayley

$$\underline{\lambda}(d : \underline{\mathbb{R}}).\ \text{SCotan}\left(\begin{array}{ccccc} & & \langle,\rangle & & \\ & \swarrow & & \searrow & \\ \langle,\rangle & & & & \langle,\rangle \\ \swarrow & \searrow & & & \swarrow \quad \searrow \\ 0 & 0 & d & & 0 \end{array}\right)$$

After Cayley

$$\underline{\lambda}(d : \underline{\mathbb{R}}).\ \text{SCotan}\left(\begin{array}{ccccc} & & \langle,\rangle & & \\ & \swarrow & & \searrow & \\ \text{id} & & & & \langle,\rangle \\ & & & & \swarrow \quad \searrow \\ & & & & (+d) \quad \text{id} \end{array}\right)$$

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 3.6, essentially reducing issue two to issue three. Afterwards, in Section 3.7, 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.

3.6 Keeping just the scalars: efficient gradient updates

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

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

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 $\text{Staged}\, c$ is really the collection of its scalars: the rest simply came from $\underline{0} : c$ and is never changed.[26] 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 $\text{Map}\, \mathbb{Z}\, \underline{\mathbb{R}}$:

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

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 $\text{Map}\, \mathbb{Z}\, \underline{\mathbb{R}}$) 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 3.5) the first complexity problem identified in Section 3.4.2 (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 Section 3.4.2 (and Section 2.2.6 on page 32) 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

[26] If c contains coproducts (sum types) as we discuss in Section 3.9, this $\underline{0} : c$ becomes dependent on the actual input to the program, copying the structure from there.

time $O\big(T \max(\log(T), \log(I)) + I \log(I)\big)$.[27] 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 3.7.

3.7 Using mutable arrays to shave off log-factors

The analysis in Section 3.4.2 showed that after the Cayley transform in Section 3.5, 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:

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

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:

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

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 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.[28]

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,

[27] 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 $O(I)$.

[28] 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].

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 chapter, 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 3.4.2, 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 3.7.1 for details). Then we can observe the following:

- The number of operations performed by $\mathbf{D}^4_c[t]$ (with the improvements from Sections 3.6 and 3.7) is only a constant factor times the number of operations performed by t, and hence in $O(T)$. This was already observed for $\mathbf{D}^2_c[t]$ in Section 3.4.2, 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 2.2.6 (page 32).

3.7.1 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`[29] 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.[30]

However, precisely because of this design, `ST` does not support parallelism. (The combination of parallelism and mutable references trivially allows nondeterministic 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.[31] Thus we can justify using `unsafePerformIO` :: `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:

$$\mathrm{Staged}\ c = \mathrm{Array}\ \underline{\mathbb{R}} \times \mathrm{Array}\ (\underline{\mathbb{R}} \multimap (\mathrm{Staged}\ c \to \mathtt{IO}\ \mathbf{1})) \times \underline{\mathbb{R}}$$

for a suitable definition of 'Array', such as `IOVector` from `vector`. To write this definition, we need to give a monoid structure on Staged $c \to$ `IO` $\mathbf{1}$; fortunately,

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

[30] 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.

[31] 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.

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.[32]

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.

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:

$$\begin{aligned}
&\texttt{alloc} &&: \mathbb{Z} \to \alpha \to \texttt{IO}\ (\text{Array}\ \alpha)\\
&\texttt{get} &&: \mathbb{Z} \to \text{Array}\ \alpha \to \texttt{IO}\ \alpha\\
&\texttt{modify} &&: \mathbb{Z} \to (\alpha \to \alpha) \to \text{Array}\ \alpha \to \texttt{IO}\ \mathbf{1}\\
&\texttt{freeze} &&: \text{Array}\ \alpha \to \texttt{IO}\ (\text{IArray}\ \alpha)\\
&(@) &&: \text{IArray}\ \alpha \to \mathbb{Z} \to \alpha
\end{aligned}$$

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 Fig. 3.10 to the version given in Figs. 3.11 and 3.12. 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 $c \to \texttt{IO}\ \mathbf{1}$ 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 Fig. 3.10: 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 3.6) 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 $\underline{\mathbb{R}}$), some more work needs to be done.

[32] $(\gg)$ :: 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.

On types:

$$\mathbf{D}_c^4[\mathbb{R}] = \mathbb{R} \times (\mathbb{Z} \times (\underline{\mathbb{R}} \multimap (\text{Staged}\, c \to \texttt{IO}\ \mathbf{1}))) \qquad \mathbf{D}_c^4[\mathbb{Z}] = \mathbb{Z} \qquad \mathbf{D}_c^4[\mathbf{1}] = \mathbf{1}$$
$$\mathbf{D}_c^4[\sigma \to \tau] = \mathbf{D}_c^4[\sigma] \to \mathbb{Z} \to \mathbf{D}_c^4[\tau] \times \mathbb{Z} \qquad \mathbf{D}_c^4[\sigma \times \tau] = \mathbf{D}_c^4[\sigma] \times \mathbf{D}_c^4[\tau]$$

On terms:

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

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

New Staged interface:

$$\text{Staged}\, c = \text{Array}\ \underline{\mathbb{R}} \times \text{Array}\ ((\underline{\mathbb{R}} \multimap (\text{Staged}\, c \to \texttt{IO}\ \mathbf{1})) \times \underline{\mathbb{R}})$$

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

$$\begin{aligned}
&\text{SCall} : \mathbb{Z} \times (\underline{\mathbb{R}} \multimap (\text{Staged}\, c \to \texttt{IO}\ \mathbf{1})) \to \underline{\mathbb{R}} \multimap (\text{Staged}\, c \to \texttt{IO}\ \mathbf{1})\\
&\text{SCall}\ \langle i, f\rangle\ a\ \langle c, m\rangle = \texttt{modify}\ i\ (\lambda\langle _, a'\rangle.\ \langle f, a + a'\rangle)\ m
\end{aligned}$$

$$\begin{aligned}
&\text{SOneHot} : \mathbb{Z} \to \underline{\mathbb{R}} \multimap (\text{Staged}\, c \to \texttt{IO}\ \mathbf{1})\\
&\text{SOneHot}\ i\ a\ \langle c, m\rangle = \texttt{modify}\ i\ (\lambda(a' : \mathbb{R}).\ a + a')\ c
\end{aligned}$$

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

Figure 3.11: Code transformation using mutable arrays, modified from Fig. 3.10. Grey parts are unchanged.

$$
\begin{aligned}
&\mathrm{Interleave}^4_\tau \quad : \tau \to \mathbb{Z} \to (\mathbf{D}^4_c[\tau] \times \mathrm{IArray}\ \underline{\mathbb{R}} \to \tau) \times \mathbb{Z}\\
&\mathrm{Interleave}^4_{\mathbb{R}} \quad = \lambda x.\ \lambda i.\ \langle\langle\langle x, \langle i, \mathrm{SOneHot}\ i\rangle\rangle, \lambda a.\ a\ @\ i\rangle, i + 1\rangle\\
&\mathrm{Interleave}^4_{\mathbf{1}} \quad = \lambda\langle\rangle.\ \lambda i.\ \langle\langle\langle\rangle, \lambda a.\ \langle\rangle\rangle, i\rangle\\
&\mathrm{Interleave}^4_{\sigma\times\tau} = \lambda\langle x, y\rangle.\ \lambda i.\ \mathbf{let}\ \langle\langle x', f_1\rangle, i'\rangle = \mathrm{Interleave}^4_\sigma\ x\ i\\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathbf{in\ let}\ \langle\langle y', f_2\rangle, i''\rangle = \mathrm{Interleave}^4_\tau\ y\ i'\\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathbf{in}\ \langle\langle\langle x', y'\rangle, \lambda a.\ \langle f_1\ a, f_2\ a\rangle\rangle, i''\rangle\\
&\mathrm{Interleave}^4_{\mathbb{Z}} \quad = \lambda n.\ \lambda i.\ \langle\langle n, \lambda a.\ n\rangle, i\rangle
\end{aligned}
$$

$\mathrm{Deinterleave}^4_\tau : \mathbf{D}^4_c[\tau] \to \tau \times (\tau \multimap (\mathrm{Staged}\, c \to \mathtt{IO}\ \mathbf{1}))$
(Same as $\mathrm{Deinterleave}^2$, except with the same monoid changes as $\mathbf{D}^4_c$ above)

$\mathrm{Wrap}^4 : (\sigma \to \tau) \rightsquigarrow (\sigma \to \tau \times (\tau \multimap \sigma))$
$\mathrm{Wrap}^4[\lambda(x : \sigma).\ t] = \lambda(x : \sigma).$
$\quad \mathbf{let}\ \langle\langle x : \mathbf{D}^4_\sigma[\sigma], \mathit{rebuild} : \mathrm{IArray}\ \underline{\mathbb{R}} \to \sigma\rangle, i\rangle = \mathrm{Interleave}^4_\sigma\ x\ 0$
$\quad \mathbf{in\ let}\ \langle y', i'\rangle = \mathbf{D}^4_\sigma[t]\ i$
$\quad \mathbf{in\ let}\ \langle y, d : \tau \multimap (\mathrm{Staged}\, \sigma \to \mathtt{IO}\ \mathbf{1})\rangle = \mathrm{Deinterleave}^4_\tau\ y'$
$\quad \mathbf{in}\ \langle y, \underline{\lambda}(z : \tau).\ \mathit{rebuild}\ (\mathtt{unsafePerformIO}$
$\qquad\qquad\qquad\qquad\qquad\qquad (\mathrm{SAlloc}\ i\ i' \gg\!= \lambda s.\ d\ z\ s \gg \mathrm{SResolve}\ i'\ s))\rangle$

Figure 3.12: Wrapper for Fig. 3.11, modified from Fig. 3.10. Grey parts are unchanged.

Firstly, 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[33] 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 $d : \tau \multimap (\text{Staged}\,\sigma \to \texttt{IO}\ \mathbf{1})$.

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 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 Fig. 3.10).

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 Fig. 3.10, we build structure information up in the form of a getter (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

[33] Implementing the $(\mathbb{Z} \to \underline{\mathbb{R}})$ in $(\mathbb{Z} \to \underline{\mathbb{R}}) \to \tau$ from Section 3.6 as IArray $\underline{\mathbb{R}}$.

remains.

On the Staged interface the transition to mutable arrays had a significant effect. The 0_{Staged} created by Wrap^3 in Section 3.5 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 3.5, 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 $\mathbb{Z} \times \underline{\mathbb{R}}$. 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 $\mathbf{D}^4_c[t]$, 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.

3.8 Was it taping all along?

In this section we first apply one more optimisation to our algorithm to make it slightly more efficient (Section 3.8.1). Afterwards, we show that defunctionalising the backpropagators (Section 3.8.2) essentially reduces the technique to classical taping approaches (Section 3.8.3).

3.8.1 Dropping the cotangent collection array

Recall that the final transformation of Section 3.7 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 $((\underline{\mathbb{R}} \multimap (\text{Staged}\, c \to \texttt{I0}\ \mathbf{1})) \times \underline{\mathbb{R}})$, using the monad-based implementation from Section 3.7.1. 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 $\langle c, m\rangle$ 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 the following as our definition:

$$\text{Staged}\, c = \text{Array}\, ((\underline{\mathbb{R}} \multimap (\text{Staged}\, c \to \mathtt{IO}\ \mathbf{1})) \times \underline{\mathbb{R}})$$

The reconstruction functions of Section 3.6 simply take the second projection of the correponding array element.

3.8.2 Defunctionalisation of backpropagators

In the core code transformation ($\mathbf{D}_c$, excluding the wrapper), all backpropagators are (now) of type $\underline{\mathbb{R}} \multimap (\text{Staged}\, c \to \mathtt{IO}\ \mathbf{1})$, and, as observed earlier in Section 3.4, these backpropagators come in only a limited number of forms:

1. The input backpropagators created in $\text{Interleave}_{\mathbb{R}}$, which are reduced to $(\underline{\lambda}(z : \underline{\mathbb{R}}).\ \mathbf{return}\ \langle\rangle)$ in Section 3.8.1;
2. $(\underline{\lambda}(z : \underline{\mathbb{R}}).\ \mathbf{return}\ \langle\rangle)$ created in $\mathbf{D}^4_c[r]$ for scalar constants r;
3. $(\underline{\lambda}(z : \underline{\mathbb{R}}).\ \text{SCall}\, d_1\ (\partial_1 op(\ldots)(z)) \circ \cdots \circ \text{SCall}\, d_n\ (\partial_n op(\ldots)(z)))$ 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

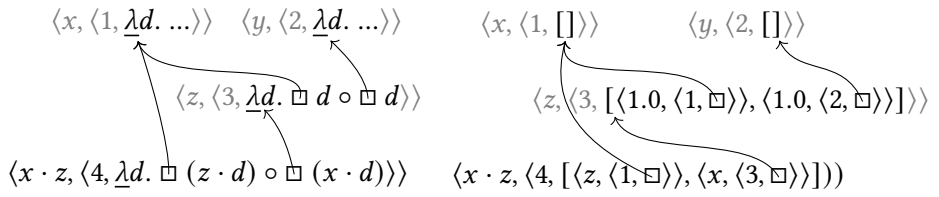
$$\partial_i op(x_1, \ldots, x_n)(z) = z \cdot \partial_i op(x_1, \ldots, x_n)(1)$$

Hence, we can defunctionalise [Reynolds 1998] and change all occurrences of the type $\underline{\mathbb{R}} \multimap (\text{Staged}\, c \to \mathtt{IO}\ \mathbf{1})$ to Contrib, where $\text{Contrib} = [\underline{\mathbb{R}} \times (\mathbb{Z} \times \text{Contrib})]$: 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 $[\langle a_1, \langle i_1, cb_1\rangle\rangle, \ldots, \langle a_n, \langle i_n, cb_n\rangle\rangle]$ of type Contrib as the backpropagator

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

For example, suppose we differentiate the running example program:

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



(a) Before defunctionalisation (b) After defunctionalisation

Figure 3.13: The sharing structure before and after defunctionalisation. SCall is elided here; in Fig. 3.13a, 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.

using the final algorithm of Section 3.7.1. The return value from the $\mathbf{D}_{\mathbb{R}\times\mathbb{R}}$-transformed code (when applied to the output from $\text{Interleave}_{\mathbb{R}\times\mathbb{R}}$) has the sharing structure shown in Fig. 3.13a. This shows how the backpropagators refer to each other in their closures.

If we perform the type replacement (Section 3.8.1) 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:

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

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

3.8.3 Was it taping all along?

After the improvements from Sections 3.8.1 and 3.8.2, what previously was a tree of (staged) calls to backpropagator functions is now a tree of Contrib values with attached IDs[34] 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.

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

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 3.7, 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.

3.9 Extending the source language

The source language (Fig. 3.4) 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 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):

$$s, t ::= \cdots \mid \text{inl}\ t \mid \text{inr}\ t \mid \textbf{case}\ s\ \textbf{of}\ \{\ \text{inl}\ x \to t_1 \mid \text{inr}\ y \to t_2\ \}$$

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:

$$\begin{aligned}
\mathbf{D}^1_c[\sigma \sqcup \tau] &= \mathbf{D}^1_c[\sigma] \sqcup \mathbf{D}^1_c[\tau] \\
\mathbf{D}^1_c[\text{inl}\ t] = \text{inl}\ \mathbf{D}^1_c[t] &\qquad \mathbf{D}^1_c[\text{inr}\ t] = \text{inr}\ \mathbf{D}^1_c[t] \\
\mathbf{D}^1_c[\textbf{case}\ s\ \textbf{of}\ \{\ \text{inl}\ x \to t_1 \mid \text{inr}\ y \to t_2\ \}] &= \\
\quad \textbf{case}\ \mathbf{D}^1_c[s]\ \textbf{of}\ \{\ \text{inl}\ x \to \mathbf{D}^1_c[t_1] &\mid \text{inr}\ y \to \mathbf{D}^1_c[t_2]\ \}
\end{aligned}$$

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

$$\frac{\Gamma \vdash t : \mathbb{R}}{\Gamma \vdash \text{sign}\ t : \text{Bool}}$$

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

If one accepts losing some of the structure-preserving nature of the transformation, it is possible to prevent redundant differentiation of t in $\mathbf{D}^1_c[\text{sign}\ t]$ by making clever substitutions in t's free variables, converting back from dual-numbers form to the plain data representation. The idea is to define functions $\varphi_\tau : \mathbf{D}^1_c[\tau] \to \tau$ and $\psi_\tau : \tau \to \mathbf{D}^1_c[\tau]$ 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 $\mathbf{D}_c^{\text{plain}}$ on terms that uses φ to convert free variables to plain values, and otherwise keeps all code plain. We then get $\mathbf{D}_c^1[\text{sign } t] = \text{sign } \mathbf{D}_c^{\text{plain}}[t]$.

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].

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

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

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 $\mathbf{D}_c^1[-]$ to all parameter types of all constructors:

$$\begin{aligned}&\textbf{data } DT_1\ \alpha = DC_1\ \alpha\ (DT_2\ \alpha) \mid DC_2\ (\mathbb{R} \times (\underline{\mathbb{R}} \multimap c))\\ &\textbf{data } DT_2\ \alpha = DC_3\ \mathbb{Z}\ (DT_1\ \alpha)\ (DT_2\ \alpha)\end{aligned}$$

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

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

Furthermore, for plain type variables, we set $\mathbf{D}_c^1[\alpha] = \alpha$.[35]

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

[35] As declaring new data types is inconvenient in Template Haskell, our proof-of-concept implementation (Section 3.11) 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 fundamental restriction.

in Fig. 3.6). 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.

3.10 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 $\langle s, t\rangle$. 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.

3.10.1 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 implementation of 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:

$$\frac{\Gamma \vdash s : \sigma \qquad \Gamma \vdash t : \tau}{\Gamma \vdash s \star t : \sigma \times \tau}$$

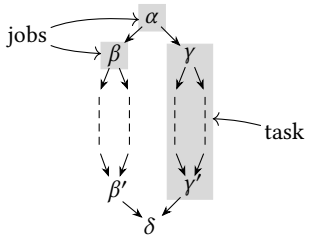


Figure 3.14: Schematic view of the operational model underlying $(\star)$.

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 $\langle v, w\rangle$ 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 3.10.3 as a compositional building block for the job graph. Figure 3.14 contains a diagrammatic representation of the preceding paragraph.

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

$$\frac{\Gamma \vdash t_1 : \tau_1 \quad \cdots \quad \Gamma \vdash t_n : \tau_n}{\Gamma \vdash \langle t_1, \ldots, t_n\rangle^\star : \tau_1 \times \ldots \times \tau_n}$$

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

$$\frac{\Gamma \vdash t : [\mathbf{1} \to \tau]}{\Gamma \vdash \star t : [\tau]}$$

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

3.10.2 Parallel IDs and their partial order

To avoid discarding the (explicit,[36] using $(\star)$) parallelism structure in the source program, we have to somehow record the dependency graph of the backpropaga-

[36]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 3.10.4 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 3.10.3).

tors (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: $\langle \alpha, \rangle \leq_c \langle \beta, i' \rangle$ iff $\alpha \leq_j \beta \wedge (\alpha \neq \beta \vee i \leq_{\mathbb{Z}} i')$. 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:

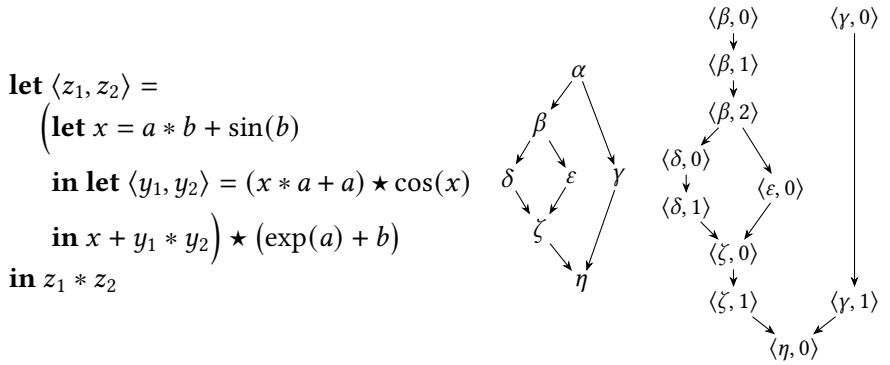
1. $\alpha \leq_j \alpha$;

2. If α forks into β and γ, then $\alpha \leq_j \beta$ and $\alpha \leq_j \gamma$;

3. If α and β join into γ, then $\alpha \leq_j \gamma$ and $\beta \leq_j \gamma$.

In Fig. 3.15, we give an example term together with graphs showing the (generators of the) partial orders $\leq_j$ on job IDs and $\leq_c$ 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 Fig. 3.15b ($\leq_j$), the arrows thus show the fork/join relationships; for Fig. 3.15c ($\leq_c$), 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 3.3 to 3.8; 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 (Fig. 3.15c) in reverse dependency order (taking advantage of task



(a) Example program (b) Job IDs ($\leq_j$) (c) Compound IDs ($\leq_c$)

Figure 3.15: 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.

parallelism), but it is actually unnecessary to construct the full graph at runtime. It is sufficient to construct the dependency graph on *job IDs* (Fig. 3.15b) 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.

3.10.3 Extending the monad

In the first optimisation that we applied to the (sequential) algorithm, namely linear factoring via staging of backpropagators (Section 3.4, Figs. 3.8 and 3.9), we modified the code transformation to produce code that runs in an state monad $\mathcal{M}\,\tau = \mathrm{ID} \to \tau \times \mathrm{ID}$, 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:

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

We saw above in Section 3.10.2 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 JID $\coloneqq \mathbb{Z}$) and write CID $\coloneqq$ JID $\times \mathbb{Z}$ for the type of compound IDs. The extended monad needs two methods:

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

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 3.10.1, 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 Fig. 3.16. 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 3.7.1.) 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 Fig. 3.14) 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 Fig. 3.8)

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

Because we now generate fresh compound IDs rather than plain integer IDs when executing primitive operations, we change $\mathbb{Z}$ to CID in $\mathbf{D}^4_c[\mathbb{R}]$:[37]

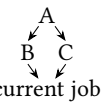
$$\mathbf{D}^5_c[\mathbb{R}] = \mathbb{R} \times (\text{CID} \times (\underline{\mathbb{R}} \multimap (\text{Staged}\, c \to \texttt{IO}\ \mathbf{1})))$$

[37] Here we work from the transformation $\mathbf{D}^4$ as described in Section 3.7.1 on monadic mutable arrays. With the defunctionalisation described in Section 3.8.2, the backpropagator would simply read 'Contrib' instead.

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

$$\text{JobDescr} = \text{History} \times \text{JID} \times \mathbb{Z}$$

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



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

Figure 3.16: 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 new rule for the parallel pairing construct is simply:

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

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 $\mathcal{M}$.

3.10.4 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 3.7, 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:

$$\text{Staged}\,c = \text{Array}\ \underline{\mathbb{R}} \times \text{Array}\ (\text{Array}\ ((\underline{\mathbb{R}} \multimap (\text{Staged}\,c \to \texttt{IO}\ \mathbf{1})) \times \underline{\mathbb{R}}))$$

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

```
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
```

Figure 3.17: Implementation of SResolve for the parallel-ready dual-numbers reverse AD algorithm. The `inParallel` function is as in Fig. 3.16.

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 Fig. 3.11). An example implementation is given in Fig. 3.17, where the first function is called *resolveTask* and the second *resolveJob*. In this way, SResolve traverses the job graph (Fig. 3.15b) 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".

3.11 Implementation

To show the practicality of our method, we provide a proof-of-concept implementation[38] of the parallel algorithm of Section 3.10, together with the improvements from Sections 3.8.1 and 3.8.2, 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 3.7.1, we realise the mutable arrays using `IOVectors`. The implementation does not incorporate the changes given in Section 3.8.3 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 3.9.

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

Benchmarks. To check that our implementation has reasonable performance in practice, we benchmark (in `bench/Main.hs`) against Kmett's Haskell `ad` library [Kmett and contributors 2021] (version 4.5.6) on a few basic functions.[39] These functions are the following (abbreviating `Double` as '`D`'):

- A single scalar multiplication of type `(D, D) -> D`;
- 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 types `data Vec3 s = Vec3 s s s` and `data Quaternion s = Quaternion s s s s`;

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

[39] 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).

	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	—	—

Table 3.1: Benchmark results of Section 3.10 + Sections 3.8.1 and 3.8.2 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).

- 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 3.10.

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 paricles being independent.

The benchmark results are summarised in Table 3.1; measurement proceeded using the `criterion`[40] library. To get statistically significant results, we measure how the timings scale with increasing n:

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

- 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 Table 3.1, 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 3.11.1).

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.

3.11.1 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 3.8.2) 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 Table 3.1 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 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 3.8.2 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×), 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 $\mathbf{D}_c[\mathbb{R}] = \mathbb{R} \times (\text{CID} \times \text{Contrib})$ 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 3.10 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 3.10 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 3.10.

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 Table 3.1 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.

3.12 Conclusions

One may ask: if the final algorithm from Section 3.7 can be argued to be "just taping" (Section 3.8.3), 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 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 Fig. 3.6, studied by Brunel et al. [2020] and [Huot et al. 2020, Section 6] — for further examples, see below in Section 3.13.1) 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 chapter further to an algorithm that is useful in practice, it is necessary to add *arrays* of scalars as a primitive type

in the transformation: many AD applications tend to involve very large arrays of scalars. This enable a significant reduction of the size of the allocated tape, and reuse much more of the structure of the original program in the reverse pass by differentiating array operations to array operations, instead of many individual scalar operations. This rather significant extension is explored in Chapter 4.

3.13 Related work

To our knowledge, the first mention of the naive dual-numbers reverse mode AD algorithm that we analyse in this chapter 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 this chapter: 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.

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 Fig. 3.6 applied to, essentially, idealised Haskell98.

3.13.1 Vectorised forward AD

There are strong parallels between our optimisations to the sequential algorithm in Sections 3.3 to 3.7 and the derivation by Krawiec et al. [2022]. Like us, 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 $\mathbf{D}^1_c[\mathbb{R}] = \mathbb{R} \times (\underline{\mathbb{R}} \multimap c)$, but instead $\mathbf{D}^1_c[\mathbb{R}] = \mathbb{R} \times c$. (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 3.8.2).

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 3.4) is encoded in VFAD's definitions of *runDelta* and *eval*, which process the record back-to-front.

Finally, our Cayley transform is encoded in the type of VFAD's *eval* function (corresponding to our SResolve), 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.[41] The linear factoring rule $f\ x + f\ y \rightsquigarrow f\ (x + y)$ 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.

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

3.13.2 Other PL literature about AD

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 2024; 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 chapter, this inversion of control can be deferred to runtime 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 3.8.3, which is also essentially taping. The AD algorithm in Dex [Paszke et al. 2021a] 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 chapter. Their implementation builds on the Adept C++ library, which implements an AD algorithm that is very different from the one discussed in this chapter. 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. 2021b; Schenck et al. 2022; Paszke et al. 2021a]. These methods are orthogonal to the ideas focussed on task parallelism that we present in this chapter.

4 Bulk Dual-Numbers Reverse AD

In Chapter 3, we took a close look at dual-numbers reverse AD in its naive form and modified it to attain the correct complexity for a reverse AD algorithm. That is to say: the output program (which computes the derivative) runs in time proportional to that of the original program, when executed on the same input. Mutable arrays were necessary to reach that goal, but limiting ourselves to pure data structures added only the expected log-factors to the complexity. However, while important, complexity-efficiency is only a means to an end: the end-goal of optimisation is *reliably fast* reverse derivatives. Our complexity analysis grants us only reliability: regardless of whether we compute derivatives slowly or quickly, there is a bound on the maximum disappointment about performance.

This leaves us the task of producing derivative programs that are actually fast. Section 3.11 showed that our prototype implementation of the dual-numbers algorithm was competitive with the `ad` Haskell library; however, this comparison target was picked for its similarity in approach, not for being the state-of-the-art in reverse AD performance. Indeed, for many practical use cases of AD, including machine learning and most statistical inference, the performance of the algorithm in Chapter 3 would be very underwhelming: there may be a bound on the maximum disappointment, but that bound is still disappointingly high.

The primary reason for this slowness is that the algorithm has administration overhead for each scalar primitive operation. Practical use cases of AD use bulk operations on large arrays of scalars, and for such bulk operations, this administration overhead dwarfs the actual computational cost of the source program. In this chapter, we study a way to improve this situation by introducing

This chapter is based on [Smeding et al. 2025], a preprint in collaboration with Mikołaj Konarski, Simon Peyton Jones and Andrew Fitzgibbon. The theory was developed and (extensively) discussed by the author of this thesis (TS) and MK with helpful input from SPJ and AF. The paper was primarily written by TS and the implementation (the `horde-ad` library) primarily by MK. This chapter stays very close to the preprint and has a rewritten introduction, corrected appendices and some typographical fixes.

first-class array support in dual-numbers reverse AD.

Effectively supporting arrays is harder than it may seem: one of the selling points of dual-numbers AD is that it should generalise to almost arbitrary programming languages, so one might think that it suffices to add a primitive Array data type and supply the user with some primitive operations on arrays. In some fashion, this does indeed suffice, but the details are important: the basic dual-numbers algorithm requires *first-order* primitive operations, so this direct approach cannot support the nice second-order array operations (SOACs) from Section 2.1.3 as fast primitive operations with fast derivatives. Furthermore, as we explain in Section 4.3 in this chapter, trying to force support for SOACs into the algorithm anyway does not work: one gets a semantically sound and complexity-efficient algorithm, but no meaningful performance improvement over the naive algorithm from Chapter 3.

The solution we propose in this chapter is to extend the basic dual-numbers algorithm to solely first-order array operations in the natural way, and then add support for one specific SOAC, build :: Int $\to$ (Int $\to \tau$) $\to$ Array τ, by rewriting it to first-order array operations using a vectorisation-like code transformation. The disadvantage of this approach is that the source language is forced to be quite limited: control flow is heavily restricted, and the language is first-order except for 'build'. In return:

1. The core AD algorithm is an immediate extension of the algorithm from Chapter 3, and as such, semantical properties as well as complexity proofs transfer with little effort; the additional code transformations that we introduce aside from the core AD algorithm are simple enough that they are easy to analyse for correctness and complexity-efficiency.

2. We gain the ability to compute usefully fast derivatives for an array programming language that is still expressive enough for many array applications.

3. We gain the ability to symbolically execute the backpropagation pass of the algorithm, meaning that the differentiated program can be compiled as a whole with no need for a tape interpreter.

As in the rest of this thesis, the algorithm in this chapter is presented as a (collection of) code transformations. However, our Haskell implementation (`horde-ad`, see Section 4.7) is based on type class instantiations, not unlike the very pretty presentation of dual-numbers forward AD in [Elliott 2009]. This type class infrastructure is further reused to implement staging (for embedding of the source language in Haskell) and symbolic execution (for full compilation of the reverse pass) in a single type class instance. An in-depth description of these topics can be found in appendices to this chapter, linked from Section 4.7.

Summary of contributions. The main contributions of this chapter are as follows:

- A relatively concise presentation of dual-numbers reverse AD (Section 4.1). Contrary to Chapter 3, we use notation from [Krawiec et al. 2022] to be consistent with the implementation; for this reason, we present it again in this chapter despite the partial overlap with Chapter 3.

- An aggressive vectorisation transform (the *bulk-operation transformation*), described in Section 4.4, that transforms the programs we want to *write* (which use element-at-a-time computation) into the programs we want to *differentiate* (which use first-order (bulk) array operations only). Time complexity is preserved by this transform, but memory use may increase.

- A code transformation that lifts the "dual-numbers" approach to AD to *dual arrays* (Section 4.5).

- An analysis on the structure of the output of the algorithm that allows us to make the differentiation algorithm fully symbolic (compile-time), eliminating all runtime overhead of the differentiation algorithm over the actual gradient code (Section 4.6).

- The type class implementation of the algorithms, detailed in Section 4.7 and thereafter.

Research is rarely ever "done", every result raising new questions to be answered; this is especially true for this chapter. There are many aspects of these algorithms that can be improved in one way or another, and the reader should see this chapter as a record of our progress. Some discussion and avenues for future work can be found at the end in Section 4.8.

4.1 Dual-numbers reverse-mode AD, again

The reverse AD algorithm used in this chapter is mostly an extension of the scalar-level algorithm described in Chapter 3 and [Krawiec et al. 2022], but we make a few small changes in its implementation details. In this section we describe the scalar-level algorithm that we build on; we lift this algorithm to arrays in Section 4.5.

4.1.1 Input language

Assume a simply-typed lambda calculus with products and ground types $\mathbb{R}$, Int with their standard primitive operations:[1]

$$\begin{aligned} \sigma, \tau &::= \mathbb{R} \mid \text{Int} \mid (\sigma, \tau) \mid \sigma \to \tau \\ s, t, u, v &::= r \mid k \mid x \mid \textbf{let } x = s \textbf{ in } t \mid (s, t) \mid \text{fst } t \mid \text{snd } t \\ &\quad \mid \lambda x.\ t \mid s\ t \mid \textbf{if } t > 0 \textbf{ then } u \textbf{ else } v \mid op(t_1, \ldots, t_n) \end{aligned}$$

where r stands for a scalar constant (i.e. of type $\mathbb{R}$), k stands for an integral constant, x is a variable reference, and op stands for any arithmetic operation such as $+_{\mathbb{R}}$, $\times_{\mathbb{R}}$, $+_{\text{Int}}$, round : $\mathbb{R} \to$ Int, etc. The word "scalar" will always refer to a *real scalar*. The let-binding construct can (for now) be recursive without problems, as long as we are only asked to differentiate input programs that do in fact terminate. We assume call-by-value evaluation semantics. Polymorphism and various further language extensions could be supported by the naive algorithm described in this section, but since the extension to arrays in the main contribution of this chapter does not easily support such extensions, we refrain from over-generalising here.

While this language is higher-order (it has full lambda abstraction and application), we require that the top-level program being differentiated (i.e. the model to be trained, or the function to be optimised, etc.) has *zeroth-order* input and output types (recall Section 2.1.1). That is to say: the input program can use function values internally as much as it likes, but we do not define what it means to take a derivative *with respect to* a function, or the derivative of a function value with respect to something else. Hence, the top-level program to differentiate must have a type of the form $\sigma_1 \to \cdots \to \sigma_n \to \tau$, where none of $\sigma_1, \ldots, \sigma_n, \tau$ mention the function arrow '$\to$'. In fact, we will assume for simplicity that the type is $\tau_{\text{in}} \to \mathbb{R}$ for zeroth-order τ_{in}. The restriction to a single input is without loss of generality because the language supports pairs; we further restrict the output to a single scalar because (1) this is by far the most common case in applications of reverse AD, (2) it simplifies the wrappers (interfaces) around the core algorithm, and (3) generalisation to more general, yet still zeroth-order, output types τ_{out} is straightforward (see Section 4.10.1).

4.1.2 Code transformation

We perform automatic differentiation with a source-to-source program transformation D, given in Fig. 4.2. A source term $t : \tau$ is transformed to a dual-number

[1] Contrary to the rest of this thesis, we use Haskell notation for the type of integers ('Int') and pairs, punning the pair type constructor with the pair value constructor. However, we do use "$\mathbb{R}$" to denote the type of floating-point numbers in use by the program, e.g. `double`, to emphasise its special function for differentiation.

$$
\begin{aligned}
D[\mathbb{R}] &= (\mathbb{R}, \text{Delta}) \\
D[\text{Int}] &= \text{Int} \\
D[(\sigma, \tau)] &= (D[\sigma], D[\tau]) \\
D[\sigma \to \tau] &= D[\sigma] \to D[\tau]
\end{aligned}
$$

(a) The type transformation.

```
data Delta = Zero
           | Input DVarName
           | Add Delta Delta
           | Scale ℝ Delta
```

(b) Defunctionalised forward derivatives.

Figure 4.1: Types for naive, scalar-level dual-numbers reverse AD; presentation after [Krawiec et al. 2022].

target term $D[t] : D[\tau]$. Every real number of type $\mathbb{R}$ in t becomes a dual number $(\mathbb{R}, \text{Delta})$ in $D[t]$; the type translation is given in Fig. 4.1a. The type Delta (defined in Fig. 4.1b) describes the derivative of the number it is paired with: see Section 4.1.3.[2]

Both transformations are quite simple, recursing over the structure of terms and types respectively. The only place where the transformation "does something" is where the program directly manipulates scalars; see the upper group of rules in Fig. 4.2. In a sense, the source program is seen as nothing more than some procedure that once in a while performs some computation on scalars, and these computations are all that we are interested in.[3] Similarly, the type transformation (Fig. 4.1a) mostly just recurses over the structure of the type, except for scalars $\mathbb{R}$, which are mapped to a *pair* of a scalar and a value of type Delta (Fig. 4.1b, explained in Section 4.1.3).

When looking at the term transformation in Fig. 4.2, one should note that the first component of an $(\mathbb{R}, \text{Delta})$ pair is always equal to the $\mathbb{R}$ value that would have been computed in the source program; these are the primals. Thus, every intermediate value computed by the source program is also computed by the transformed program, and in the same order.

Larger target language. The code transformation $D[-]$ from Fig. 4.2 maps from the small input language from Section 4.1.1 into a larger output language that also includes the Delta type and its constructors. This is typical for automatic differentiation: a particular term language is not necessarily closed under differentiation in the first place (e.g. $\times_{\mathbb{R}}$ begets addition; 'log' begets division; 'arcsin' begets 'sqrt'), and reverse-mode differentiation makes this worse (where e.g. product constructors beget product projections and vice-versa). Furthermore, with dual-numbers reverse-mode AD, which looks a lot like tracing AD (Section 2.2.8,

[2] In Section 3.8.2, the equivalent of Delta is called Contrib, which can be seen as a combination of Zero, Add and Scale. Delta can also be seen as a reification of the data flow graph (Section 2.2.4) of the normal function evaluation; however, it lacks a way to encode sharing, which will be fixed in Section 4.1.5 with Share.

[3] This is what allows the algorithm to accept higher-order code without any work.

$$x_1 : \tau_1, \ldots, x_n : \tau_n \vdash t : \tau \quad \rightsquigarrow \quad x_1 : D[\tau_1], \ldots, x_n : D[\tau_n] \vdash D[t] : D[\tau]$$

— Primitive operations on real numbers include derivative computations:

$$\begin{aligned}
D[r] &= (r, \mathsf{Zero}) \\
D[\sin t] &= \mathbf{let}\ (x, d) = D[t] \\
&\quad \mathbf{in}\ (\sin x, \mathsf{Scale}\ (\cos x)\ d) \\
D[t_1 +_{\mathbb{R}} t_2] &= \mathbf{let}\ (x_1, d_1) = D[t_1]; (x_2, d_2) = D[t_2] \\
&\quad \mathbf{in}\ (x_1 +_{\mathbb{R}} x_2, \mathsf{Add}\ d_1\ d_2) \\
D[t_1 \times_{\mathbb{R}} t_2] &= \mathbf{let}\ (x_1, d_1) = D[t_1]; (x_2, d_2) = D[t_2] \\
&\quad \mathbf{in}\ (x_1 \times_{\mathbb{R}} x_2, \mathsf{Add}\ (\mathsf{Scale}\ x_2\ d_1)\ (\mathsf{Scale}\ x_1\ d_2))
\end{aligned}$$

— In general for $t_1 : \mathbb{R}, \ldots, t_n : \mathbb{R}$ and $op(t_1, \ldots, t_n) : \mathbb{R}$:

$$\begin{aligned}
D[op(t_1, \ldots, t_n)] = \mathbf{let}\ &(x_1, d_1) = D[t_1]; \ldots; (x_n, d_n) = D[t_n] \\
\mathbf{in}\ &(op(x_1, \ldots, x_n) \\
&, \mathsf{Add}\ (\mathsf{Scale}\ \left(\tfrac{\partial op(x_1, \ldots, x_n)}{x_1}\right)\ d_1) \\
&\qquad (\mathsf{Add}\ \ldots\ (\mathsf{Scale}\ \left(\tfrac{\partial op(x_1, \ldots, x_n)}{x_n}\right)\ d_n)))
\end{aligned}$$

— Everything else is structure-preserving:

$$\begin{aligned}
D[x] &= x & D[k : \mathrm{Int}] &= k \\
D[(s, t)] &= (D[s], D[t]) & D[s \times_{\mathrm{Int}} t] &= D[s] \times_{\mathrm{Int}} D[t] \\
D[\mathrm{fst}\ t] &= \mathrm{fst}\ D[t] & D[\mathbf{let}\ x = s\ \mathbf{in}\ t] &= \mathbf{let}\ x = D[s]\ \mathbf{in}\ D[t] \\
D[\mathrm{snd}\ t] &= \mathrm{snd}\ D[t] & D[\mathbf{if}\ t > 0\ \mathbf{then}\ u\ \mathbf{else}\ v] &= \\
D[\lambda x.\ t] &= \lambda x.\ D[t] & & \mathbf{if}\ \mathrm{fst}\ D[t] > 0\ \mathbf{then}\ D[u]\ \mathbf{else}\ D[v] \\
D[s\ t] &= D[s]\ D[t]
\end{aligned}$$

Figure 4.2: Rules of the code transformation for dual-numbers reverse AD, complementing Fig. 4.1.

Chapter 3), there must be some way to represent this trace (i.e. the Delta data type), which is something that did not exist in the source program.

In the rest of this chapter, the differentiating code transformation will continue to map into a larger language; its type system and semantics will always be clear from context. This means that the algorithm in this chapter cannot be used directly for higher derivatives via iterated differentiation; we leave computation of higher derivatives by computing a longer Taylor series prefix (see e.g. [Huot et al. 2022]) to future work.

4.1.3 Delta terms

In this section we focus on the Delta value that is the second component of each dual number. As we shall see, it represents the derivative of the first (primal) component. First, we recall some notation from Section 2.2. Given a function[4] $f : \mathbb{R}^n \to \mathbb{R}$, the vector:

$$\nabla f\ v = \left(\frac{\partial f}{\partial v_1}(v), \ldots, \frac{\partial f}{\partial v_n}(v)\right)$$

is the vector of partial derivatives of f at input $v : \mathbb{R}^n$, with respect to each of its n inputs $v = (v_1 \ldots v_n)$. That is, the i'th component of the vector $\nabla f\ v$ describes how the output varies as you vary v_i. We furthermore have $\nabla f\ v = (Df)^\top\ v\ 1$.

The goal of reverse AD is to compute $\nabla f\ v$, but we start with just an elaborate implementation of forward AD. In particular, we have two steps: first, we use the transformation in Section 4.1.2 to transform the function $f : \mathbb{R}^n \to \mathbb{R}$ into $f' : (\mathbb{R}, \mathsf{Delta})^n \to (\mathbb{R}, \mathsf{Delta})$ (representing the forward derivative of f), and then we use an *evaluator* to convert the Delta to an actual derivative. The correctness criterion for the transformation is:

$$\begin{array}{ll} \text{if} & f'\ ((v_1, \mathsf{Input}\ 1), \ldots, (v_n, \mathsf{Input}\ n)) = (r, d) \\ \text{then} & r = f(v_1, \ldots, v_n) \\ \text{and} & \mathit{eval}_0\ d\ \varepsilon = Df\ v\ \varepsilon = (\nabla_v f) \odot \varepsilon = ((Df)^\top\ v\ 1) \odot \varepsilon \end{array}$$

using the Df and $(Df)^\top$ notation from Section 2.2. Here, $\mathsf{Input}\ 1, \ldots, \mathsf{Input}\ n$ are values in the Delta type (Fig. 4.1b); each is paired with its corresponding input value, $v = (v_1 \ldots v_n)$. The primal result of f' is r, and will be equal to $f(v)$. The second component of the result is d : Delta, a data structure that describes, or represents, the derivative of f. More precisely, we can *evaluate* d in a direction described by ε to get the forward derivative of f in the direction ε.

[4] For now we consider only functions of type $\mathbb{R}^n \to \mathbb{R}$. Generalising to richer input and output types is straightforward, but adds a lot of notational clutter.

The evaluation function $eval_0$ could not be more straightforward: it just interprets Add as addition, Scale as multiplication, and so on:

$$\begin{aligned}
&eval_0 :: \mathsf{Delta} \to \mathbb{R}^n \to \mathbb{R} \\
&eval_0\ \mathsf{Zero} &&\varepsilon = 0 \\
&eval_0\ (\mathsf{Input}\ i) &&\varepsilon = (\text{the } i\text{'th component of } \varepsilon) \\
&eval_0\ (\mathsf{Add}\ d_1\ d_2) &&\varepsilon = eval_0\ d_1\ \varepsilon + eval_0\ d_2\ \varepsilon \\
&eval_0\ (\mathsf{Scale}\ r\ d) &&\varepsilon = r \cdot eval_0\ d\ \varepsilon
\end{aligned}$$

Thus, the Delta data type is really a little straight-line programming language in which we express the forward derivative of the function.

4.1.4 Efficient gradients 1: a single pass

However, we want to compute a gradient, not just the forward derivative. How can we do that, given $f' : (\mathbb{R}, \mathsf{Delta})^n \to (\mathbb{R}, \mathsf{Delta})$? One obvious way is to use the evaluator ($eval_0$) n times, like this:

$$\begin{aligned}
\nabla_v f = &(eval_0\ d\ (1, 0, \ldots, 0), \ldots, eval_0\ d\ (0, 0, \ldots, 1)) \\
&\text{where} \\
&\quad (r, d) = f'\ ((v_1, \mathsf{Input}\ 1), \ldots, (v_n, \mathsf{Input}\ n))
\end{aligned} \tag{4.1}$$

But, following [Krawiec et al. 2022], a natural optimisation is to write a new evaluator $eval_1$ that computes those n results simultaneously, thus:

$$\begin{aligned}
&eval_1 :: \mathsf{Delta} \to \mathbb{R}^n \\
&eval_1\ \mathsf{Zero} &&= (0, 0, \ldots, 0) \\
&eval_1\ (\mathsf{Input}\ i) &&= (\text{one-hot vector with 1 at position } i) \\
&eval_1\ (\mathsf{Add}\ d_1\ d_2) &&= eval_1\ d_1 + eval_1\ d_2 \\
&eval_1\ (\mathsf{Scale}\ r\ d) &&= r \cdot eval_1\ d
\end{aligned}$$

where $(+)$ and $(\cdot)$ operate elementwise. With this formulation we only need to call $eval_1$ once, but it creates, scales, and adds, many n-vectors, which is not efficient if n is large.[5] Fortunately, it is not difficult to transform $eval_1$ into an evaluator that transforms a *single* n-vector, and does so in a completely single-threaded way, amenable to mutable in-place updates. To do so, we apply Cayley transformation (also known as the "difference list trick"):

$$\begin{aligned}
&eval_2 :: \mathsf{Delta} \to \mathbb{R}^n \to \mathbb{R}^n \\
&eval_2\ \mathsf{Zero} &&= \mathsf{id} \\
&eval_2\ (\mathsf{Input}\ name) &&= \lambda tg.\ (tg \text{ with 1 added to position } name) \\
&eval_2\ (\mathsf{Add}\ d_1\ d_2) &&= eval_2\ d_2 \circ eval_2\ d_1 \\
&eval_2\ (\mathsf{Scale}\ r\ d) &&= (r\cdot) \circ eval_2\ d
\end{aligned}$$

[5] Ending the story at $eval_1$ (and ignoring the sharing issues, see Section 4.1.5) would yield an $O(nT)$ reverse AD algorithm, where T is the runtime of f and n is the input size. Optimal is $O(T)$.

We have $eval_2\ d\ g_0 = eval_1\ d + g_0$.

While this has improved the efficiency of the first three cases, the case for Scale still does $O(n)$ work to multiply by r elementwise, where we would like it to be $O(1)$ instead. To address this, we "push" the scale factors r inside so that at Input nodes, we do not add just '1', but the value it would have been after all the scaling factors have been applied:

$$\begin{aligned}
&eval_3 :: \mathbb{R} \to \mathsf{Delta} \to \mathbb{R}^n \to \mathbb{R}^n \\
&eval_3\ c\ \mathsf{Zero} &&= \mathsf{id} \\
&eval_3\ c\ (\mathsf{Input}\ name) &&= \lambda tg.\ (tg \text{ with } c \text{ added to position } name) \\
&eval_3\ c\ (\mathsf{Add}\ d_1\ d_2) &&= eval_3\ c\ d_2 \circ eval_3\ c\ d_1 \\
&eval_3\ c\ (\mathsf{Scale}\ r\ d) &&= eval_3\ (c \cdot r)\ d
\end{aligned}$$

We have $eval_2\ d\ v = eval_3\ 1\ d\ v$. The accumulating parameter is called c for cotangent, as it is the incoming cotangent for this node in the data flow graph.

An interesting observation is that in rewriting $eval_2$ to $eval_3$, the accumulation order of the derivative values has flipped from forward order to reverse order. Indeed, for the following example Delta term:

$$\mathsf{Scale}\ r_1\ (\mathsf{Scale}\ r_2\ (\ldots\ (\mathsf{Scale}\ r_n\ (\mathsf{Input}\ n))))$$

$eval_2$ would compute $r_1 \cdot (r_2 \cdot (\ldots \cdot (r_n \cdot 1)))$, whereas $eval_3\ 1$ would compute $(((r_1 \cdot r_2) \cdot \ldots) \cdot r_n) \cdot 1$. This reversal is expected in a *reverse-mode AD* algorithm, which we need to be able to calculate gradients efficiently.

While $eval_3$ improves computational complexity significantly over $eval_1$, we do still have a second problem: lost sharing.

4.1.5 Efficient gradients 2: respecting sharing

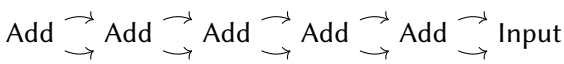
Consider the program $x : \mathbb{R} \vdash P_1 : \mathbb{R}$ given by:

$$\begin{aligned}
&\textbf{let}\ x_1 = x +_{\mathbb{R}} x \\
&\quad\ x_2 = x_1 +_{\mathbb{R}} x_1 \\
&\quad\ \vdots \\
&\quad\ x_n = x_{n-1} +_{\mathbb{R}} x_{n-1} \\
&\textbf{in}\ x_n
\end{aligned}$$

The code transformation will transform this to $x : (\mathbb{R}, \mathsf{Delta}) \vdash D[P_1] : (\mathbb{R}, \mathsf{Delta})$:

$$\begin{aligned}
&\textbf{let}\ x_1 = \textbf{let}\ (y_1, d_1) = x; (y_2, d_2) = x\ \textbf{in}\ (y_1 +_{\mathbb{R}} y_2, \mathsf{Add}\ d_1\ d_2) \\
&\quad\ x_2 = \textbf{let}\ (y_1, d_1) = x_1; (y_2, d_2) = x_1\ \textbf{in}\ (y_1 +_{\mathbb{R}} y_2, \mathsf{Add}\ d_1\ d_2) \\
&\quad\ \vdots \\
&\quad\ x_n = \textbf{let}\ (y_1, d_1) = x_{n-1}; (y_2, d_2) = x_{n-1}\ \textbf{in}\ (y_1 +_{\mathbb{R}} y_2, \mathsf{Add}\ d_1\ d_2) \\
&\textbf{in}\ x_n
\end{aligned}$$

The in-memory representation of the Delta term in x_n looks like this:



but a plain interpreter of a Delta term cannot see this sharing, and thus evaluation ($eval_0$ for forward AD, and $eval_3$ for reverse AD) will be *exponential* in n. This is clearly unacceptable.

We follow Chapter 3 (see Section 3.4) in solving this problem. The idea is to "simply" make the sharing visible: we give every fragment of a Delta term that may later be shared a unique name (implemented as a numeric ID). Concretely, we add an additional constructor 'Share ID Delta' to Delta:

```
data Delta = Zero | Input DVarName | Add Delta Delta | Scale ℝ Delta
           | Share ID Delta
type ID = Int    — for semantic clarity
```

and we give unique IDs to all potentially shareable Delta (sub)terms by wrapping them in a Share constructor with that unique ID. We have two invariants on those IDs:

Invariant 1. In a Delta term 'Share i d', all IDs appearing inside d are strictly smaller than i.

Invariant 2. Given two Delta terms 'Share i d_1' and 'Share j d_2', if $i = j$ then d_1 and d_2 live at the same address in memory.

In practice, the converse of invariant 2 is also true (which is useful for efficiency), but for soundness we need only the invariant as stated.

Together, these invariants ensure that the sharing structure is soundly represented and acyclic (i.e., the Delta term represents a directed acyclic graph (DAG)), and furthermore that it is helpful for ensuring that *eval* can avoid evaluating any part of the Delta term more than once. The trick is that the new *eval* will backpropagate through Delta subterms in strict decreasing order of ID. Whenever it encounters a Share node, *eval* will save the c value to be backpropagated into that part of the Delta term; if the same node is encountered multiple times, the c values are added. Backpropagation into a node is resumed only when it is next-in-line: its ID is the highest among the Delta subterms still to process. See the discussion of the sharing-aware evaluator in Section 4.1.6 for how this is implemented.

Lifting to monadic code. To be able to generate these unique IDs, we lift the right-hand side of $D[-]$ to monadic code in a state monad with a single Int as state.

$$x_1 : \tau_1, \ldots, x_n : \tau_n \vdash t : \tau \quad \rightsquigarrow \quad x_1 : D[\tau_1], \ldots, x_n : D[\tau_n] \vdash D[t] : \mathrm{IdGen}\ D[\tau]$$

— Updated type transformation:

$$D[\mathbb{R}] = (\mathbb{R}, \mathrm{Delta}) \qquad D[(\sigma, \tau)] = (D[\sigma], D[\tau])$$
$$D[\mathrm{Int}] = \mathrm{Int} \qquad D[\sigma \to \tau] = D[\sigma] \to \mathrm{IdGen}\ D[\tau]$$

— The ID generation monad:

$$\textbf{newtype}\ \mathrm{IdGen}\ a = \mathrm{IdGen}\ (\mathrm{State}\ \mathrm{Int}\ a)$$
$$\mathrm{genID} :: \mathrm{IdGen}\ \mathrm{ID} \qquad \textit{— Recall that ID} = \mathrm{Int}.$$

— Primitive operations on real numbers:

$$\begin{aligned}
D[r] &= \textbf{return}\ (r, \mathrm{Zero}) \\
D[\sin t] &= \textbf{do}\ (x, d) \leftarrow D[t] \\
&\qquad\quad id \leftarrow \mathrm{genID} \\
&\qquad\quad \textbf{return}\ (\sin x, \mathrm{Share}\ id\ (\mathrm{Scale}\ (\cos x)\ d)) \\
D[t_1 +_{\mathbb{R}} t_2] &= \textbf{do}\ (x_1, d_1) \leftarrow D[t_1]; (x_2, d_2) \leftarrow D[t_2] \\
&\qquad\quad id \leftarrow \mathrm{genID} \\
&\qquad\quad \textbf{return}\ (x_1 +_{\mathbb{R}} x_2, \mathrm{Share}\ id\ (\mathrm{Add}\ d_1\ d_2)) \\
D[t_1 \times_{\mathbb{R}} t_2] &= \textbf{do}\ (x_1, d_1) \leftarrow D[t_1]; (x_2, d_2) \leftarrow D[t_2] \\
&\qquad\quad id \leftarrow \mathrm{genID} \\
&\qquad\quad \textbf{return}\ (x_1 \times_{\mathbb{R}} x_2, \mathrm{Share}\ id\ (\mathrm{Add}\ (\mathrm{Scale}\ x_2\ d_1)\ (\mathrm{Scale}\ x_1\ d_2)))
\end{aligned}$$

etc. the other primitive operations on $\mathbb{R}$

— Other operations are monadically lifted:

$$\begin{aligned}
D[k] &= \textbf{return}\ k \\
D[x] &= \textbf{return}\ x \\
D[\textbf{let}\ x = s\ \textbf{in}\ t] &= \textbf{do}\ x \leftarrow D[s]; D[t] \\
D[(s, t)] &= \textbf{do}\ x \leftarrow D[s]; y \leftarrow D[t]; \textbf{return}\ (x, y) \\
D[\mathrm{fst}\ t] &= \textbf{do}\ x \leftarrow D[t]; \textbf{return}\ (\mathrm{fst}\ x) \\
D[\mathrm{snd}\ t] &= \textbf{do}\ x \leftarrow D[t]; \textbf{return}\ (\mathrm{snd}\ x) \\
D[\lambda x.\ t] &= \textbf{return}\ (\lambda x.\ D[t]) \\
D[s\ t] &= \textbf{do}\ f \leftarrow D[s]; x \leftarrow D[t]; f\ x \\
D[\textbf{if}\ t_1\ \textbf{then}\ t_2\ \textbf{else}\ t_3] &= \textbf{do}\ x \leftarrow D[t_1]; \textbf{if}\ x\ \textbf{then}\ D[t_2]\ \textbf{else}\ D[t_3] \\
D[s \times_{\mathrm{Int}} t] &= \textbf{do}\ x \leftarrow D[s]; y \leftarrow D[t]; \textbf{return}\ (x \times_{\mathrm{Int}} y)
\end{aligned}$$

etc. the other primitive operations on Int *and* Bool

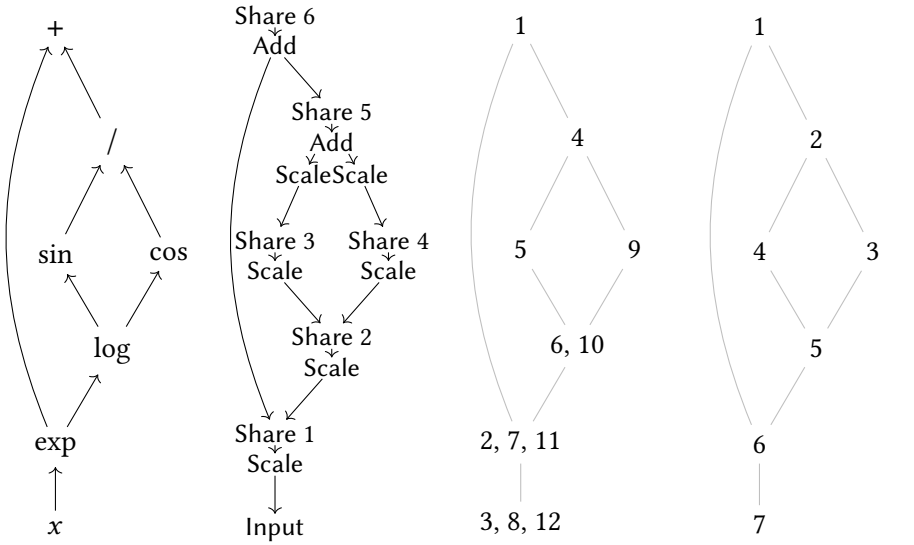
Figure 4.3: Updated rules for the dual-numbers reverse AD code transformation to properly handle sharing. Compare Fig. 4.2; the added text (apart from the lifting to monadic code) is highlighted in red.

The updated code transformation is given in Fig. 4.3. Note that for all language constructs except primitive operations, this monadic lifting is done very systematically, as functional programmers (especially in Haskell) are well used to. The only wrinkle is that because our language so far supports user-written functions, and the bodies of those functions get differentiated too, those differentiated functions also become effectful. This results in the updated rule $D[\sigma \to \tau] = D[\sigma] \to \text{IdGen}\ D[\tau]$ in Fig. 4.3, as well as the fact that in $D[s\ t]$, there is no '**return**' around the result of the call $f\ x$.

For primitive operations, we use 'genID', the (only) monad method, to generate unique, strictly monotonically increasing ID values — the strict monotonicity allows us to preserve invariant 1. Note that we only give an ID to the full Delta value returned by the code for each primitive operation; for example, there is no Share node wrapping the Scale constructors in the Delta value for $(\times_{\mathbb{R}})$. We can leave these out because these Scale nodes can never be shared: they are used exactly once, namely in the containing Add, which itself *does* have an ID. Furthermore, we elide the Share node around Zero in the right-hand side of $D[r]$, because while that Zero may well be used multiple times, $eval_3$ of Zero is very cheap and not worth deduplicating.

This scheme does generally result in "too many" IDs: contrary to the somewhat contrived P_1 from the beginning of this subsection, in practice far from all Delta terms that we give a unique ID to will actually be shared. But we certainly have *enough* Share nodes: every Delta node that could benefit from being evaluated only once, gets a unique ID.

This way of recording sharing inside Delta terms using IDs is quite different from the usual way of notating shared subterms in an expression language: let-bindings.[6] In contrast with let-bindings, where the shared term is available only in a limited scope (the 'in' part of the let-binding), with our Share-based approach, the shared term is "available" everywhere apart from inside the shared term itself. For this reason, we call this Share-based approach *global sharing*; this idea will be used again in Section 4.6.



(a) Source program (b) Delta term (c) Naive visit order (d) Sharing-aware order

Figure 4.4: $eval_3$ of Section 4.1.4 visits Delta nodes as often as there are paths to them; the sharing-aware evaluator of Section 4.1.6, $reversePass_4$, visits Share-wrapped nodes only once. From left to right: **(a)** the source expression **let** $y = \exp x$ **in let** $z = \log y$ **in** $y + \frac{\sin z}{\cos z}$ with input x (arrows in the direction of execution); **(b)** the resulting Delta term (arrows pointing to subterms; IDs assume the sin was executed before the cos); **(c)** the order in which $eval_3$ visits the nodes of (b) if Share nodes were removed; **(d)** the order in which $reversePass_4$ visits the nodes of (b).

The grey lines in (c) and (d) are only to visualise the relation to (a) and (b).

4.1.6 Evaluator

The evaluator that we present now improves over $eval_3$ from Section 4.1.4 by using the IDs in Share nodes to visit every Delta node only once (excepting Zero and Input, which take $O(1)$ time to evaluate anyway). The resulting evaluation order is shown on a simple example program in Fig. 4.4. Note that $eval_3$ would have visited the 'exp' and x nodes in Fig. 4.4a three times, whereas the new evaluator visits them only once.

The improved evaluator does not simply traverse the Delta term depth-first, but instead in a mix of breadth-first and depth-first traversal. Accordingly, the evaluator is restructured into multiple functions:

1. $reversePass_4$, a (non-recursive) wrapping function that initialises the backpropagation process and calls $backprop_4$.
2. $backprop_4$, which loops over the found IDs from highest to lowest, evaluating the Delta fragment for each in turn using $eval_4$.
3. $eval_4$, which interprets a Delta fragment under a Share constructor, stopping at any contained Share constructors and deferring their recursive traversal until $backprop_4$ decides that their time has come.

The result is shown in Fig. 4.5. To be able to delay recursing below Share nodes, we need more storage in our evaluation state: in addition to the sparse gradient accumulator in 'grad' (which was implicitly a mutable $\mathbb{R}^n$ before in $eval_3$, but which we make explicitly sparse now using Map DVarName $\mathbb{R}$), we need storage to save the Delta trees that we still need to visit, as well as their accumulated cotangents. These are stored, respectively, in 'dfrag' and 'accum' in the evaluation state ES. The main backpropagation loop is $backprop_4$, which repeatedly chooses the largest encountered-but-yet-unvisited ID, takes it out of the 'dfrag' and 'accum' maps, runs $eval_4$ on this Delta fragment, and then continues with the next unvisited ID. Evaluating a Delta fragment proceeds exactly as before in $eval_3$ (Section 4.1.4), except that on reaching a Share constructor, evaluation does not recurse but instead saves the contained Delta tree, as well as the cotangent c to be backpropagated into that tree, in the evaluation state. The cotangent is added to the value already in the state, if any. (This is the add operation in the

[6] Krawiec et al. [2022] instead choose to make Delta a proper, traditional term language with let-bindings; this requires a more complicated monad. In short, the monad becomes additionally a writer monad of (ID, Delta) pairs, and Delta is augmented with two constructors: Var ID and Let ID Delta. In $D[-]$ for primitive operations, instead of simply naming the returned Delta term d, the pair (id, d) is emitted in the writer monad and we return only Var id. Before evaluation, the writer log is stacked in chronological order (which is also increasing ID order) and set as a stack of Let bindings on top of the final Delta term from the program. We use our global sharing approach instead because we will need it in Section 4.6.

```
data ES = ES    — evaluation state
  { grad :: Map DVarName ℝ    — input cotangents: will collect final gradient
  , dfrag :: Map ID Delta     — delta fragments
  , accum :: Map ID ℝ }       — accumulated node cotangents

reversePass4 :: ℝ → Delta → Map DVarName ℝ
reversePass4 c d = grad (backprop4 (eval4 c d (ES {} {} {})))

backprop4 :: ES → ES
backprop4 s = case Map.maxViewWithKey (accum s) of
                Just ((i, c), acc') →
                  let d = dfrag s Map.! i
                      s' = eval4 c d (s { accum = acc'
                                        , dfrag = Map.delete i (dfrag s) })
                  in backprop4 s'
                Nothing → s

eval4 :: ℝ → Delta → ES → ES
eval4 c Zero        = id
eval4 c (Input v)   = λs. s { grad = Map.insertWith (+) v c (grad s) }
eval4 c (Add d1 d2) = eval4 c d2 ∘ eval4 c d1
eval4 c (Scale r d) = eval4 (c · r) d
eval4 c (Share i d) = λs. s { dfrag = Map.insert i d (dfrag s)
                            , accum = Map.insertWith (+) i c (accum s) }
```

Figure 4.5: The Delta evaluator that handles internal sharing in Delta terms and has the right time complexity, apart from logarithmic factors due to use of Map.

```
wrapper :: AST ℝ    — with one free variable: x : τ_in
        → τ_in → ℝ → τ_in
wrapper t inp ctg =
  let inp' = named_τ_in inp
      (_, d) = runIdGen (let x = inp' in D[t]) 0    — Provide D[t]'s free
      grad = reversePass_4 ctg d                    variable x
  in reconstruct_τ_in grad inp'
— Using the following functions:
runIdGen :: IdGen a → Int → a
named_τ_in :: τ_in → D[τ_in]    — for zeroth-order τ_in
    — e.g. named_((ℝ,ℝ),ℝ) ((7, 3.1), 8) =
              (((7, Input 0), (3.1, Input 1)), (8, Input 2))
reconstruct_τ_in :: Map DVarName ℝ → D[τ_in] → τ_in    — for zeroth-order τ_in
    — e.g. reconstruct_((ℝ,ℝ),ℝ) grad (((7, Input 0), (3.1, Input 1)), (8, Input 2)) =
              ((grad ! 0, grad ! 1), grad ! 2)
```

Figure 4.6: The wrapper for the dual-numbers reverse AD transformation of Fig. 4.3. The implementations of 'named' and 'reconstruct' are straightforward but somewhat verbose. Technically 'reconstruct' does not need the $D[\tau_{\text{in}}]$ argument for our language, but we add it for generality, because it would have been necessary had we included e.g. coproducts (sum types) in our type system.

reverse derivative that comes from sharing in the source program, according to the mantra: "sharing in the primal becomes addition in the dual".) When there are no other unvisited subtrees with higher IDs, $backprop_4$ will take these values out of the state and backpropagate the summed cotangent contributions down into the saved Delta tree by invoking $eval_4$ again.

At the top level, $reversePass_4$ initialises the process by starting backpropagation on a state produced by evaluating the topmost fragment of the full Delta term resulting from the forward pass. The final gradient is simply the 'grad' field of the state after backpropagation is complete.

In the remainder of the chapter, plain use of '*reversePass*', '*backprop*' or '*eval*' will refer to their 4th version in Fig. 4.5.

4.1.7 Wrapper

As in [Krawiec et al. 2022] and Chapter 3, we add a wrapper around the algorithm to give it a useful API, and to make explicit what that API *is*, precisely.

An implementation sketch is given in Fig. 4.6. Our first input is a term t

satisfying the typing $x : \tau_{\text{in}} \vdash t : \mathbb{R}$ for some zeroth-order τ_{in}. (Generalisation of the output type $\mathbb{R}$ would require running the reverse pass once for each output scalar; Section 4.10.1 shows how to do this for the final algorithm of Section 4.6.) We transform this term to $x : D[\tau_{\text{in}}] \vdash D[t] : (\mathbb{R}, \text{Delta})$, which we can run once we have a $D[\tau_{\text{in}}]$. To obtain such a $D[\tau_{\text{in}}]$, we take a τ_{in}-typed input (*inp*, the point to differentiate at) and pair up each scalar in that structure with an Input node with a unique DVarName. This yields *inp*′ of type $D[\tau_{\text{in}}]$ (recall Fig. 4.1a). Thus we can now evaluate the transformed term at the processed input point to obtain the function result ($\mathbb{R}$) and its forward derivative (Delta). The function result could be returned as well, but is ignored in Fig. 4.6.

Then we call $reversePass_4$ (Fig. 4.5) with the initial $\mathbb{R}$ cotangent (which we get from the user, typically '1') to obtain the gradient in a 'Map DVarName $\mathbb{R}$'. Finally, replacing all scalar-Input pairs in *inp*′ by their corresponding scalar from the gradient map (*grad*), we obtain the actual gradient of type τ_{in}, which we return.

It is worth noting that to differentiate the same program at multiple input points, almost everything needs to be repeated *for each such point*: only the transformed term $D[t]$ can be cached. There is never a good opportunity to optimise the calculations of the reverse pass, because the output of the Delta interpretation process (in $reversePass_4$) is already a *gradient*, not a gradient-computing *program* that we can still compile and optimise. We address this efficiency problem as well (in Section 4.6), after we have improved support for arrays.

Interpretation of derivatives on discrete types. In the world of mathematics, if we are given a function

$$f : \mathbb{Z}^{k_1} \times \mathbb{R}^{n_1} \to \mathbb{Z}^{k_2} \times \mathbb{R}^{n_2}$$

and are asked for its Jacobian (the total derivative), we seem to have to define what it means to take a partial derivative of an integer value with respect to another. This makes little sense: taking the classical limit definition of a derivative at its word (interpreted in the discrete topology of the integers), such a "partial derivative" would necessarily be identically 0, so it contains no information.

Because of this, our transformation joins Delta terms only to scalar values, not to discrete values in the input and output of the function to differentiate. Mathematically, this corresponds to not computing Jf, but instead ignoring the discrete outputs of f and reinterpreting its discrete inputs as global constants:

$$\tilde{f} : \mathbb{R}^{n_1} \to \mathbb{R}^{n_2}$$

and then computing $J\tilde{f} : \mathbb{R}^{n_1} \to \mathbb{R}^{n_2 \times n_1}$, a function that produces the Jacobian matrix of $\tilde{f}$ at any given input point.

Our choice of not computing derivatives for integral values means that the wrapper in Fig. 4.6 has nothing sensible to return for integers in τ_{in}; 'reconstruct' has to choose something, and we leave unspecified what it chooses. (Reasonable options include "zero" and "the input".) To reflect the undefinedness of a derivative with respect to an integer value, we could give 'wrapper' a more precise type:

$$\text{wrapper} :: \text{AST}\,\mathbb{R} \to \tau_{\text{in}} \to \mathbb{R} \to T[\tau_{\text{in}}]$$

where T is a type function that maps a type to its type of tangents:[7]

$$T[\mathbb{R}] = \mathbb{R} \qquad T[\text{Int}] = () \qquad T[(\sigma, \tau)] = (T[\sigma], T[\tau])$$

With this typing, 'wrapper' does not need to return nonsensical values. For convenience, however, we let it return the full τ_{in} here.

In the rest of this chapter, we continue to think about derivatives and Jacobians as if this T is implicitly applied, both in the input and in the output (if the part of the transformation in question supports non-trivial output types). For example, we say that the function round : $\mathbb{R} \to \text{Int}$ has trivial derivative (it contains no information), because its Jacobian, being an element of $\mathbb{R}^{0\times 1}$, is empty.

4.2 A language with arrays: the core language

The reverse AD algorithm set out in Section 4.1 works fine on functions that manipulate scalars, but real programs that AD is used on typically work with *arrays* of scalars. This includes machine learning applications such as neural networks, but also probabilistic programming on larger statistical models, most optimisation applications, etc. Thus, to really count as an AD system, we ought to support arrays. In what way?

From Section 2.1 we know that functional array languages come in multiple flavours, chief among which first-order (Section 2.1.2) and second-order ones (Section 2.1.3). Second-order array languages are strictly more expressive than first-order ones: the first-order combinators can be defined in terms of the second-order ones, but not the other way round. Furthermore, providing second-order combinators with free array indexing to the programmer allows them to write more understandable code; this has been eloquently argued by Paszke et al. [2021a] in their design philosophy of the Dex language. For example, if one knows that

[7] Properly we would need *cotangents* here, but the types of tangents and cotangents coincide for products of scalars and discrete types. See also Chapter 5 and Section 5.2.

the following code is possible for naive matrix multiplication:[8]

$$
\begin{aligned}
&\text{matmat} :: \text{Array } 2\ \mathbb{R} \to \text{Array } 2\ \mathbb{R} \to \text{Array } 2\ \mathbb{R}\\
&\text{matmat } a\ b =\\
&\quad \textbf{let } [k, m] = \text{shape } a\\
&\qquad\ [\_, n\ ] = \text{shape } b\\
&\quad \textbf{in } \text{build } k\ (\lambda i.\ \text{build } n\ (\lambda j.\\
&\qquad\qquad \text{sum } (\text{build } m\ (\lambda p.\ a\ !\ [i, p] * b\ !\ [p, j]))))
\end{aligned}
$$

then one would certainly not be satisfied with having to write something like the following:

$$
\begin{aligned}
&\text{matmatFirstOrder} :: \text{Array } 2\ \mathbb{R} \to \text{Array } 2\ \mathbb{R} \to \text{Array } 2\ \mathbb{R}\\
&\text{matmatFirstOrder } a\ b =\\
&\quad \textbf{let } [k, m] = \text{shape } a\\
&\qquad\ [\_, n\ ] = \text{shape } b\\
&\quad \textbf{in } \text{sumInner } (\text{transpose } (\text{replicate } n\ a) * \text{replicate } k\ (\text{transpose } b))
\end{aligned}
$$

Both versions of 'matmat' do the same thing, but the first is significantly easier to understand and to get correct.

4.2.1 Our core language

Thus, we want our array language (the "core" language — used for both input and output of the differentiation algorithm) to be in second-order style as much as possible.[9] Unfortunately, it turns out that a generic fold/reduction operation is very difficult to support in a dual-numbers reverse AD framework if one desires fast gradient code, as we do. Thus, while we do have 'build' and all the other operations derivable from it (such as 'map'), we have to make do with specific reductions like 'sum' and 'maximum'.[10] For conciseness, we include only 'sum' explicitly in the language, because adding support for other specific reductions is simple and requires little more than writing down their derivative.

The syntax of the core language is given in Fig. 4.7, and the type system and typing rules (some abbreviated as pseudo-type-signatures) are given in Figs. 4.8 and 4.9. Some notes to clarify parts of Figs. 4.7 to 4.9 that might be unfamiliar or unclear:

[8] The 'shape' function returns the list of sizes of the dimensions of the array, outermost first.

[9] On the other hand, it turns out that because first-order (bulk) operations are much better for efficient reverse AD using the dual-numbers framework, we end up converting 'build'-code into first-order operations anyway in Section 4.4.

[10] The implementation (Section 4.7) does experimentally support a general fold operation, restricted to closed combination functions.

$s, t, u, v ::= c$	(constant literal tensors, e.g. $[[3, 1.2, 17], [-5, 0.4, 1]]$)
$\mid x \mid \textbf{let}\ x = u\ \textbf{in}\ v$	(variables and binding)
$\mid \mathsf{cond}\ t\ u\ v$	(strict conditionals)
$\mid op\ u\ v \mid op\ t$	(broadcasted (elementwise) binary and unary ops.)
$\mid \mathsf{index}\ t\ ix$	(index at a multidimensional position)
$\mid \mathsf{sumOuter}\ t$	(reduce along the outermost dimension)
$\mid \mathsf{gather}\ sh\ t\ (\lambda is.\ ix)$	(backward permutation; Section 4.2.2)
$\mid \mathsf{scatter}\ sh\ t\ (\lambda is.\ ix)$	(forward permutation onto zeros; see Section 4.2.2)
$\mid [t_1, \ldots, t_n]$	(combine equal-shaped arrays into 1 extra dimension)
$\mid \mathsf{replicate}\ k\ t$	(add an outermost dimension of size k by replication)
$\mid \mathsf{tr}_{k_1,\ldots,k_n}\ t$	(generalised transposition; n = rank of t, $k_1, \ldots, k_n$ must be a permutation of $0, \ldots, n-1$)
$\mid \mathsf{reshape}\ sh\ t$	(product of sh = product of (shape of t))
$\mid \mathsf{build1}\ k\ (\lambda i.\ t)$	(construct a new array elementwise)
$k ::= 0 \mid 1 \mid 2 \mid \ldots$	(a constant (i.e. static) natural)
$sh ::= [] \mid k ::: sh$	(a constant shape; $[k_1, \ldots, k_n] \stackrel{\text{def}}{=} k_1 ::: \ldots ::: k_n ::: []$)
$ix ::= [] \mid t ::: ix$	(a dynamic index; $[t_1, \ldots, t_n] \stackrel{\text{def}}{=} t_1 ::: \ldots ::: t_n ::: []$)
$is ::= [] \mid x ::: is$	(index variables; $[x_1, \ldots, x_n] \stackrel{\text{def}}{=} x_1 ::: \ldots ::: x_n ::: []$)

Figure 4.7: The grammar of the core language. We variously use other variable names than x in expressions, especially "i" for embedded variables of type 'Array [] Int'.

- We use the word *rank* to denote the number of dimensions of an array, and by extension, for array-typed terms, the number of dimensions of their output.

- The language is shape-typed: the *shape* (the list of all dimensions' sizes) of an array is reflected on the type-level. This results in typing that is stronger than most other array languages. For example, a 3-by-2 array of scalars, where "2" is the size of the inner dimension, would have type Array $[3, 2]\ \mathbb{R}$.

- Tuples and nested arrays are unsupported:[11] every expression is of array type, and the only arrays are multidimensional arrays of *element types* (denoted by ρ in Fig. 4.8). Hence, all arrays are *regular*: there are no jagged arrays. What one might expect to be scalar subexpressions are really zero-dimensional arrays in our language; see, for example, the "$\Gamma \vdash$ *ix* is an *n*-dim. index" judgement in Fig. 4.8, as well as its use in e.g. the rule for index in the same figure.

- '*op*' stands for an arbitrary unary or binary primitive arithmetic or comparison operator on scalars; we consider these to automatically broadcast to arrays of equal shapes. Thus, $s + t$ is valid if s and t are terms producing arrays of the same shapes, and computes their elementwise sum. We lump all of these together in a single syntactic element because the differences are immaterial in most of the algorithms in this chapter.

- 'tr' is a generalised array transposition: if a is a 4-dimensional array with shape $[5, 3, 6, 9]$ (with 5 being the outermost dimension and 9 the innermost), then '$\mathrm{tr}_{3,0,1,2}\ a$' is a 4-dimensional array with shape $[9, 5, 3, 6]$. Also see the typing rule for tr in Fig. 4.9.

- 'sumOuter' reduces elementwise along the *outermost* dimension. This is to be dual with replicate, making its derivative rule in Section 4.5 more elegant; but note that an e.g. inner-dimension sum can be recreated from sumOuter by combining it with some transpositions.

- The expressions that make up an index expression *ix* cannot have sharing between them in our grammar. This is relevant in the index mapping functions passed to gather and scatter (for their semantics, see below in Section 4.2.2). This is for simplicity of presentation and not a fundamental limitation.

[11] The implementation fully supports pairs, and provides some support for nested arrays through `https://hackage.haskell.org/package/ox-arrays`. As these extensions do not produce interesting algorithmic problems, we exclude them in this chapter for simplicity.

Types:

$$\rho ::= \mathbb{R} \mid \text{Int} \mid \text{Bool}$$
$$\sigma, \tau ::= \text{Array } sh\ \rho \qquad (sh \text{ is a list of non-negative integers})$$

Typing rules:

$$\frac{\Gamma \vdash t_1 : \text{Array } [] \text{ Int} \quad \ldots \quad \Gamma \vdash t_n : \text{Array } [] \text{ Int}}{\Gamma \vdash [t_1, \ldots, t_n] \text{ is an } n\text{-dim. index}} \qquad \boxed{\Gamma \vdash ix \text{ is an } n\text{-dim. index}}$$

$$\frac{}{\mathbb{R} \text{ numeric}} \qquad \frac{}{\text{Int numeric}} \qquad \boxed{\rho \text{ numeric}}$$

$$\boxed{\Gamma \vdash t : \tau}$$

$$\frac{c \text{ an array of shape } sh \text{ filled with } \rho s}{\Gamma \vdash c : \text{Array } sh\ \rho} \qquad \frac{x : \text{Array } sh\ \rho \in \Gamma}{\Gamma \vdash x : \text{Array } sh\ \rho}$$

$$\frac{\Gamma \vdash u : \text{Array } sh_1\ \rho_1 \qquad \Gamma, x : \text{Array } sh_1\ \rho_1 \vdash v : \text{Array } sh_2\ \rho_2}{\Gamma \vdash \textbf{let } x = u \textbf{ in } v : \text{Array } sh_2\ \rho_2}$$

$$\text{cond} : \text{Array } 0 \text{ Bool} \to \text{Array } sh\ \rho \to \text{Array } sh\ \rho \to \text{Array } sh\ \rho$$
$$op : \text{Array } sh\ \rho \to \text{Array } sh\ \rho \to \text{Array } sh\ \rho \qquad \text{(binary arithmetic ops.)}$$
$$op : \text{Array } sh\ \rho \to \text{Array } sh\ \rho \to \text{Array } sh \text{ Bool} \qquad \text{(binary comparison ops.)}$$
$$op : \text{Array } sh\ \rho \to \text{Array } sh\ \rho \qquad \text{(unary arithmetic operations)}$$
$$\text{sumOuter} : \text{Array } (k ::: sh)\ \rho \to \text{Array } sh\ \rho \qquad \text{(for numeric } \rho \text{ (i.e. } \mathbb{R}, \text{Int}))$$

$$\frac{\Gamma \vdash t : \text{Array } [k_1, \ldots, k_n]\ \rho \qquad \Gamma \vdash ix \text{ is an } m\text{-dim. index} \qquad m \le n}{\Gamma \vdash \text{index } t\ ix : \text{Array } [k_{m+1}, \ldots, k_n]\ \rho}$$

Figure 4.8: Typing rules for the core language. Continued in Fig. 4.9.

$$\frac{\Gamma \vdash t : \text{Array } [k_1, \ldots, k_{m_2}, k_{m_2+1}, \ldots, k_n]\ \rho \qquad \Gamma, i_1, \ldots, i_{m_1} : \text{Array } [] \text{ Int} \vdash ix \text{ is an } m_2\text{-dim. index} \qquad m_1, m_2 \leq n}{\Gamma \vdash \text{gather } [k'_1, \ldots, k'_{m_1}, k_{m_2+1}, \ldots, k_n]\ t\ (\lambda[i_1, \ldots, i_{m_1}].\ ix) : \text{Array } [k'_1, \ldots, k'_{m_1}, k_{m_2+1}, \ldots, k_n]\ \rho}$$

$$\frac{\Gamma \vdash t : \text{Array } [k_1, \ldots, k_{m_1}, k_{m_1+1}, \ldots, k_n]\ \rho \qquad \rho \text{ numeric} \qquad \Gamma, i_1, \ldots, i_{m_1} : \text{Array } [] \text{ Int} \vdash ix \text{ is an } m_2\text{-dim. index} \qquad m_1, m_2 \leq n}{\Gamma \vdash \text{scatter } [k'_1, \ldots, k'_{m_2}, k_{m_1+1}, \ldots, k_n]\ t\ (\lambda[i_1, \ldots, i_{m_1}].\ ix) : \text{Array } [k'_1, \ldots, k'_{m_2}, k_{m_1+1}, \ldots, k_n]\ \rho}$$

$$\frac{\Gamma \vdash t_1 : \text{Array } sh\ \rho \quad \ldots \quad \Gamma \vdash t_n : \text{Array } sh\ \rho}{\Gamma \vdash [t_1, \ldots, t_n] : \text{Array } (n ::: sh)\ \rho}$$

$$\frac{k \text{ a constant integer} \geq 0 \qquad \Gamma \vdash t : \text{Array } sh\ \rho}{\Gamma \vdash \text{replicate } k\ t : \text{Array } (k ::: sh)\ \rho}$$

$$\frac{j_1, \ldots, j_m \text{ is a permutation of } 0, \ldots, m-1 \qquad \Gamma \vdash t : \text{Array } [k_1, \ldots, k_n]\ \rho \qquad m \leq n}{\Gamma \vdash \text{tr}_{j_1, \ldots, j_m}\ t : \text{Array } [k_{j_1+1}, \ldots, k_{j_m+1}, k_{m+1}, \ldots, k_n]\ \rho}$$

$$\frac{\Gamma \vdash t : \text{Array } [k_1, \ldots, k_m]\ \rho \qquad \prod_{i=1}^{m} k_i = \prod_{i=1}^{n} k'_i}{\Gamma \vdash \text{reshape } [k'_1, \ldots, k'_n]\ t : \text{Array } [k'_1, \ldots, k'_n]\ \rho}$$

$$\frac{k \text{ a constant integer} \geq 0 \qquad \Gamma, i : \text{Array } [] \text{ Int} \vdash t : \text{Array } sh\ \rho}{\Gamma \vdash \text{build1 } k\ (\lambda i.\ t) : \text{Array } (k ::: sh)\ \rho}$$

Figure 4.9: Continuation of Fig. 4.8.

There are a number of peculiarities and restrictions in this core language that result from the "bulk-operation transformation" that we will apply to the program in Section 4.4, before the actual differentiation. The most important ones are:

- Statically known array shapes only: the size of array dimensions is not allowed to depend on intermediate values computed earlier in the program. If one wants to rerun a differentiated program on differently sized arrays, one has to re-differentiate and re-compile the program.
- Conditionals have strict semantics, sometimes called *selections*: 'cond t u v' first evaluates t, u and v, and subsequently returns the value of either the second or the third argument depending on the value of the first.
- Finally, the language does not support separate top-level functions; all must be a single expression (with possible internal let-bindings, of course).

Very roughly, these restrictions exist because we want to be able to eliminate build1 from the program by "vectorising" it into other array operations. After introducing the bulk-operation transformation, we discuss these restrictions again in Section 4.4.2.

4.2.2 Semantics of gather and scatter

Gather. As can be inferred from its typing rule in Fig. 4.9, the 'build1' primitive in the language constructs a $(k + 1)$-dimensional array given a function that maps a single index to a k-dimensional array. Using build1, we can create a multidimensional build operation as a notational shorthand:

$$\begin{aligned}&\mathsf{build}\ [k_1, \ldots, k_n]\ (\lambda[i_1, \ldots, i_n].\ t) \stackrel{\text{def}}{=} \\ &\quad \mathsf{build1}\ k_1\ (\lambda i_1.\ \ldots (\mathsf{build1}\ k_n\ (\lambda i_n.\ t))\ldots)\end{aligned}$$

Then, the 'gather' primitive is really just a specialisation of this 'build':

$$\mathsf{gather}\ sh\ a\ (\lambda is.\ t) = \mathsf{build}\ sh\ (\lambda is.\ \mathsf{index}\ a\ t))$$

Gather can be said to be "batched array indexing". We need gather explicitly in the language, despite it being expressible using build1, in order to properly represent the output of the bulk-operation transformation. This will be discussed in more detail in the following sections.

Scatter. The 'scatter' operation is the dual of 'gather', and is included in the language not only because it is necessary for histogram-like operations (which cannot be otherwise expressed using the rest of the core language), but

also because it forms the reverse derivative of 'gather' (see Section 4.5.3). In 'scatter $sh\ t\ (\lambda[i_1, \ldots, i_n].\ ix)$', the argument sh gives the shape of the result of the operation, t is the array of input values to be scattered, and the function determines *where* the elements of t are to be written in the output array. Multiple values sent to the same location are added with $(+)$. For example, using single-dimensional arrays only, the following program (writing flooring integer division as a binary operator (div)):

$$\text{scatter}\ [6]\ [1,2,3,4,5,6,7,8,9]\ (\lambda[i].\ [i\ \text{div}\ 2])$$

returns the array $[3, 7, 11, 15, 9, 0]$. This result is computed as follows:

- Indices 0 and 1 are both sent to $0\ \text{div}\ 2 = 1\ \text{div}\ 2 = 0$, thus the values 1 and 2 are added together to yield 3.
- The last value in the source array (9) is sent to index $8\ \text{div}\ 2 = 4$, and it is the only element sent to this position; hence the result has 9 at index 4.
- No element is sent to index 5 of the output, hence the result is zero.

4.3 Naive extension to arrays: Unsuccessful

Now that we have an array language to differentiate, let us try to extend the basic dual-numbers reverse AD algorithm from Section 4.1 to our core language from Section 4.2.1 in the "obvious" way, and see what goes wrong. The problems that arise will inform the changes and optimisations that we make, eventually resulting in the final algorithm.[12]

4.3.1 Scalar dual numbers: The Delta explosion problem

The promise of dual-numbers AD is that it is extensible to almost any imaginable program construct by just adding more rules to the code transformation $D[-]$ that map over the new constructs in a structure-preserving way. Let us attempt this for arrays, seeing an array as little more than a very large product type. On the type level, we get:

$$D[\text{Array}\ sh\ \rho] = \text{Array}\ sh\ D[\rho]$$

but what of the array operations? In Fig. 4.3 we had:

$$D[s\ t] = \mathbf{do}\ f \leftarrow D[s]; x \leftarrow D[t]; f\ x$$

[12]The designs in this section were already suggested in [Krawiec et al. 2022, §8.3]; we discuss them in more detail and improve upon them.

and our array operations look like functions, so ostensibly we get something like this:[13]

$$\begin{aligned} D[\text{build1 } k\ (\lambda i.\ t)] &= \textbf{do } f \leftarrow D[\text{build1}]; n \leftarrow D[k]; f\ n\ (\lambda i.\ D[t]) \\ D[\text{index } t\ [t_1, \ldots, t_n]] &= \textbf{do } f \leftarrow D[\text{index}]; a \leftarrow D[t] \\ &\qquad\qquad i_1 \leftarrow D[t_1]; \ldots; i_n \leftarrow D[t_n] \\ &\qquad\qquad f\ a\ [i_1, \ldots, i_n] \\ D[\text{sumOuter } t] &= \textbf{do } f \leftarrow D[\text{sumOuter}]; a \leftarrow D[t]; f\ a \end{aligned}$$

But then what are $D[\text{build1}]$, $D[\text{index}]$, $D[\text{sumOuter}]$, etc.?

It is worth noting that build1 and index on the one hand, and sumOuter on the other hand, are quite different when it comes to differentiation; let us look at sumOuter on scalars[14] first. Let us limit ourselves to the 2-dimensional case for notational simplicity (read "plane" or "subarray" instead of "row" for 3 or higher dimensions, respectively). Then the normal operation of sumOuter is to sum the rows of a matrix elementwise, producing a single row.

$$\begin{matrix}[[1, & 2, & 3], \\ \ +, & \ +, & \ + \\ [4, & 5, & 6], \\ \ +, & \ +, & \ + \\ [7, & 8, & 9]]\end{matrix} = [12,\ 15,\ 18]$$

For the derivative of sumOuter, instead of an array of scalars we get an array of dual numbers that we need to add:

$$\begin{matrix}[[(1, d_1), & (2, d_2), & (3, d_3)], \\ \ +, & \ +, & \ + \\ [(4, d_4), & (5, d_5), & (6, d_6)], \\ \ +, & \ +, & \ + \\ [(7, d_7), & (8, d_8), & (9, d_9)]]\end{matrix} = \begin{matrix}[(12, \text{Share } \_\ (\text{Add } (\text{Add } d_1\ d_2)\ d_3)), \\ \ (15, \text{Share } \_\ (\text{Add } (\text{Add } d_4\ d_5)\ d_6)), \\ \ (18, \text{Share } \_\ (\text{Add } (\text{Add } d_7\ d_8)\ d_9))]\end{matrix} \tag{4.2}$$

Note that there must be Share nodes around the Delta terms in the result because they may be used multiple times; the '_'s stand for unique generated IDs. For the time being, let us assume that there is some function 'DsumOuter' in the target language that does precisely this: take an n-dimensional array of dual numbers and return an $(n-1)$-dimensional array of dual numbers by summing elementwise along the outer dimension. With this, we get:

$$D[\text{sumOuter } t] = \textbf{do } a \leftarrow D[t]; \text{DsumOuter } a$$

Note that 'DsumOuter' is a monadic operation because it needs to generate unique IDs for the Share nodes in Eq. (4.2).

[13] We are abusing syntax here: e.g. 'build1 k $(\lambda i.\ t)$' is a term, but technically 'build1' is not.

[14] The type system also allows summing arrays of integers, but $D[\text{Int}] = \text{Int}$, so we simply get $D[\text{sumOuter}_{\text{Int}}] = \textbf{return } \text{sumOuter}_{\text{Int}}$.

Observe that we needed to examine the performed computation, differentiate it, and represent the differentiated result again as a program. For build1 and index, the story is quite different. This is because these operations are both parametrically polymorphic in the element type of the arrays they produce (build1) or consume (index). They just "move elements around", and are sufficiently uncaring when the array element type changes from scalar to non-scalar, or (indeed!) to dual numbers. In this, build1 and index are no different than 'fst', lambda-abstraction, etc. from Fig. 4.3, which we could just differentiate to themselves (modulo monadic lifting). And indeed, it turns out that doing the same to build1 and index works equally well, as long as we handle the fact that any functions passed *to* build1 are monadically lifted too. We obtain the following derivatives:

$$\begin{aligned} D[\mathsf{build1}\ k\ (\lambda i.\ t)] &= \mathbf{sequence}\ (\mathsf{build1}\ k\ (\lambda i.\ D[t])) \\ D[\mathsf{index}\ t\ [t_1, \ldots, t_n]] &= \mathbf{do}\ a \leftarrow D[t]; i_1 \leftarrow D[t_1]; \ldots, i_n \leftarrow D[t_n] \\ &\qquad \mathbf{return}\ (\mathsf{index}\ a\ [i_1, \ldots, i_n]) \end{aligned} \tag{4.3}$$

where '**sequence**' has type Array sh (IdGen ρ) $\to$ IdGen (Array sh ρ) and evaluates all monadic computations in the array, producing an array of results.

Apart from this wrinkle of having to propagate the effects, we indeed maintain the structure-preserving quality of the transformation. Because these array operations do not act on scalars directly, they are just "structure", and are thus preserved by the algorithm.[15]

While these definitions are correct, and the complexity requirements are met, the resulting performance is very unsatisfactory. Consider a simple dot product operation, expressed by the term t_{dot} with two free variables, a and b:

$$\begin{aligned} &a : \text{Array}\ [n]\ \mathbb{R}, b : \text{Array}\ [n]\ \mathbb{R} \\ &\vdash t_{\text{dot}} = \mathsf{sumOuter}\ (\mathsf{build1}\ n\ (\lambda[i].\ \mathsf{index}\ a\ [i] \times_{\mathbb{R}} \mathsf{index}\ b\ [i])) : \text{Array}\ []\ \mathbb{R} \end{aligned}$$

Of course, this implementation is suboptimal: a dedicated loop can easily be more than 4× faster than this program, in part by eliminating the materialised intermediate array of products. Its naive derivative, however, is far worse still, and exemplifies the problem with the array operation derivatives in Eq. (4.3) just above. After transformation to dual numbers (and some basic simplifications for

[15] Proving correctness of these derivatives is somewhat subtle. The index operation is algebraically linear, and since a (forward) derivative is the best linear approximation of a function, the forward derivative of a linear function is just itself. (This idea was explored further by Elsman et al. [2022].) For 'build1', one can build confidence by expanding into individual scalar operations; a full proof requires an induction argument (using logical relations) following [Huot et al. 2020; Lucatelli Nunes and Vákár 2024].

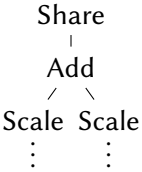
readability), the program looks as follows:

$$
\begin{aligned}
D[t_{\text{dot}}] = \textbf{do } c \leftarrow \textbf{sequence } (&\text{build1 } n \ (\lambda i.\ \textbf{do} \\
&\quad \textbf{let } (x_1, d_1) = \text{index } a \ [i]; (x_2, d_2) = \text{index } b \ [i] \\
&\quad id \leftarrow \text{genID} \\
&\quad \textbf{return } (x_1 \times_{\mathbb{R}} x_2 \\
&\qquad\qquad , \text{Share } id \ (\text{Add } (\text{Scale } x_2 \ d_1) \ (\text{Scale } x_1 \ d_2))))) \\
&\text{DsumOuter } c
\end{aligned}
$$

Consider what $D[t_{\text{dot}}]$ does: it will build *a Delta term the size of the input array*, containing (when counting carefully) $5n + 1$ Delta data constructors when given inputs of length n.[16] Aside from using far too much memory (thus also destroying memory locality), this whole tree will need to be interpreted, node by node, in the reverse pass, which allocates even more memory to hold various administrative data about all the $2n$ inputs plus the $n + 1$ Share nodes. Furthermore, there is little hope of vectorising the (actually very structured) multiplications and additions in the reverse pass.

Meanwhile, a proper implementation of the reverse derivative of a dot product simply consists of two (very efficiently implementable) multiplications of a scalar with a vector. Thus, even if this approach of adding arrays to our language is very neat, simple and extensible, it will not fly in practice.

Doing better. Let us call the problem of allocating (and interpreting) far too many Delta nodes the *Delta explosion* problem. To fix this problem, the first thing we notice is that while we do create a tremendous number of Delta nodes, many of them look very similar! Indeed, *all* of the runs of the lambda in 'build1' in $D[t_{\text{dot}}]$ return a Delta subgraph with the exact same structure:



where the dots stand for the Delta terms of the scalars index a i and index b i. The subgraphs differ only in the ID in the Share node and the scalars in the Scale nodes. It would be good if we can represent the Delta term computed by $D[t_{\text{dot}}]$ more compactly: from the viewpoint of the arrays, are there really only a few (bulk) operations being done, and each of those has a simple derivative.

In essence, the Delta term produced by a given differentiated program is really a *trace* of the primitive operations executed by the program:[17] with 'build1' and

[16] $4n$ from the lambda to build1 and $n + 1$ for the Adds and the outer Share in DsumOuter.

[17] The "entries" in this trace contain only the partial derivatives of the operations executed, not the operations themselves.

'sumOuter' as-is, this trace is too fine-grained for efficient differentiation. By itself, the fact that dual-numbers reverse AD generates a trace is unsurprising and already noted in Section 3.8. However, what what we really want is a trace of size $O(\#\text{array operations})$ instead of $O(\#\text{scalar operations})$, so that we still know the array structure of the source program when we start computing its gradient.

4.3.2 Dual arrays: A step in the right direction

To accomplish this reduction of the trace (i.e. Delta term) size, we have to teach the algorithm to consider arrays differentiable objects in and of themselves. To that effect, we replace the naive rule $D[\text{Array } sh\ \rho] = \text{Array } sh\ D[\rho]$ with the following:

$$D[\text{Array } sh\ \rho] = (\text{Array } sh\ \rho, \text{Delta } sh)$$

With this rule, Delta must now also be able to represent forward derivatives of *array* computations. Hence, we give it a type parameter *sh*: where 'Delta' described the forward derivative of a computation of type $\mathbb{R}$ before, this 'Delta *sh*' describes the forward derivative of a computation of type Array *sh* $\mathbb{R}$. Its base is the same as before (Zero, Input, Add, Scale, Share), but we add more constructors for each of the primitive array operations:

```
data Delta sh where
  Zero      :: Delta sh
  Input     :: DVarName → Delta sh
  Add       :: Delta sh → Delta sh → Delta sh
  Scale     :: Array sh ℝ → Delta sh → Delta sh
  Share     :: ID → Delta k → Delta k
  — Most array operations get a dedicated constructor:
  Index     :: Delta [k_1, ..., k_n] → Ix m → Delta [k_{m+1}, ..., k_n]
  SumOuter  :: Delta (k ::: sh) → Delta sh
  Gather    :: Delta [k_1, ..., k_{m_2}, k_{m_2+1}, ..., k_n] → (Ix m_1 → Ix m_2)
               → Delta [k'_1, ..., k'_{m_1}, k_{m_2+1}, ..., k_n]
  Replicate :: Delta sh → Delta (k ::: sh)
  — etc., others elided
```

where Ix m is an m-dimensional index, i.e. simply m integers:

```
data Ix k where
  IZ    :: Ix 0
  (:::) :: Int → Ix k → Ix (k + 1)
```

Indeed, because we use Delta to represent the forward derivative of programs in our language, Delta must certainly be able to (somehow) represent the forward derivatives of all *primitive operations* in our language. So far, for primitive

arithmetic operations with at most one scalar output, we could make do with Zero, Scale and Add for this purpose, because (forward) derivatives are linear and all linear functions $\mathbb{R}^n \to \mathbb{R}$ are simply linear combinations. For example, the forward derivative of $\lambda x\ y.\ x \times_{\mathbb{R}} y$ at inputs x, y is $\lambda dx\ dy.\ y \times_{\mathbb{R}} dx + x \times_{\mathbb{R}} dy$, which is written $\lambda dx\ dy$. Add (Scale y dx) (Scale x dy) in the Delta language (compare Fig. 4.3).

For more general linear functions, however, this normal form (a linear combination) generalises to a matrix: for each individual scalar in the output of the linear function, we could give an Add/Scale/Zero expression in terms of the operation's inputs. The resulting array of scale factors (for each output with respect to each input) is precisely the Jacobian matrix of the operation that this linear function is the derivative of. The size of this matrix is (#scalars in output) · (#scalars in input), and while typically sparse, the sparsity pattern is heavily dependent on the specific operation (e.g. gather, replicate, sumOuter). So a good sparse representation would need to be a sum type over all the primitive operations, storing in each case just the information necessary to reconstruct the full Jacobian.[18]

It turns out that a very good sparse representation of the Jacobian of a particular array operation is simply a program that computes the forward derivative (how its output changes in response to a particular change to its inputs; the *total derivative*). The Delta constructors that we add for the primitive array operations, such as Index, SumOuter, etc. in **data** Delta *sh* above, are precisely that — and their semantics is what one expects, just like the semantics of Add, Scale, etc. whas precisely what one expects under *eval*.

Forward derivative of a linear function is itself. Given the function $f = \lambda x\ y.\ 2x + 5y$, some input x, y, and some small change $\Delta x, \Delta y$ to that input, how much does $f\ (x + \Delta x)\ (y + \Delta y)$ differ from $f\ x\ y$? Well, $2\Delta x + 5\Delta y$, surely, because f is a linear function (a vector space homomorphism). The forward derivative of f at some input x, y is simply f itself. This holds for all linear functions, and surprisingly many useful functions are linear: as a special case, as long as a function just rearranges and/or adds values from its input, it is certainly linear, and this is in fact true for all array operations in our core language (Fig. 4.7) except for the arithmetic operations *op* and our higher-order operation build1 (because its lambda argument may be non-linear). Thus, for example, the Delta constructor corresponding to index (i.e. its forward derivative) has the same semantics as the index operation itself, and is hence simply called 'Index'.

[18] It would be grossly inefficient to just materialise all those Jacobians densely.

4.3.3 The Delta explosion problem again

So the linear array operations are relatively straightforward, and it turns out that because our primitive arithmetic operations on arrays are elementwise, their derivative Delta terms are essentially the same as we wrote for the scalar algorithm in Fig. 4.3. So what about the one remaining operation, build1?

$$D[\text{build1 } k\ (\lambda i.\ t)] = ??$$

Say for simplicity that t : Array [] $\mathbb{R}$. Then the source term is of type Array $[k]\ \mathbb{R}$, hence the '??' should be of type $D[\text{Array } [k]\ \mathbb{R}] = (\text{Array } [k]\ \mathbb{R}, \text{Delta } [k])$. Regardless of how exactly we compute it, the second component of '??' (call it d) should be a Delta term that describes the forward derivative of the whole build1 operation, and furthermore the number of nodes in this Delta term d should be much less than k (if we are to fix the Delta explosion problem).

However, regardless of how we extend the Delta data type, surely d depends on t, and furthermore it depends on the execution paths that each *individual* execution of t took for each index i. Indeed, the source term's derivative depends on those execution paths, and d should express precisely that derivative. The most we can do (apart from non-compositionally handling special cases) is something like this:

```
data Delta sh where
  — ... Zero, Input, etc. as before
  Build1 :: Array [k] (Delta sh) → Delta (k ::: sh)
```

The array of Deltas records the forward derivatives of the lambda for each index i of the build operation. While this extension of Delta surely allows us to write down a derivative of build1, it does not solve anything: we have as many Delta nodes as before, only organised differently.[19]

The source of this problem is that there is still elementwise scalar computation in the program (in particular, inside build1), and this scalar computation must be differentiated faithfully to a Delta term.[20] In contrast, *first-order* array operations, e.g. elementwise arithmetic operators such as '$(\vec{+}) :: \text{Array } sh\,\mathbb{R} \to \text{Array } sh\,\mathbb{R} \to \text{Array } sh\,\mathbb{R}$' but also other bulk array operations that we already have such as sumOuter, can be differentiated without any trouble as primitive operations in the language. The trouble comes from user-written scalar-level code that is executed many times.

Before we fix the problem with build1, let us investigate a second problem that has silently appeared: the *one-hot problem*.

[19] Where previously we had $D[\text{Array } sh\ \mathbb{R}] = \text{Array } sh\ (\mathbb{R}, \text{Delta})$, we now essentially have $D[\text{Array } sh\ \mathbb{R}] = (\text{Array } sh\ \mathbb{R}, \text{Array } sh\ \text{Delta})$; that Array sh Delta is just wrapped in a Build1 constructor.

[20] Specifically, computation on *scalars*: computation on individual *integers*, like in 'gather', is perfectly fine because differentiation does not touch that.

4.3.4 The one-hot problem

As it turns out, the move to dual arrays not only has left the Delta explosion problem unsolved (in build1), it also creates a new problem: the derivative for index, while easily written down, has very bad performance. This problem is not yet visible in Delta; a priori, the Index constructor of Delta that we gave in Section 4.3.2 seems quite reasonable:

```
data Delta sh where
  — ... etc.
  Index :: Delta [k_1, ..., k_n] → Ix m → Delta [k_{m+1}, ..., k_n]
  — ... etc.
```

However, while the forward derivative of indexing looks (and is) innocuous, its reverse derivative is problematic: the transposed interpretation of Index (as *eval* will need to implement) has to take the cotangent of a single array element and produce a cotangent for the array that the element was projected from. This produced array cotangent will be a *one-hot array*: in case the indexed array is single-dimensional, this looks like $[0, \ldots, 0, d, 0, \ldots, 0]$, where d is the incoming cotangent for Index and its position in the one-hot vector is the original projection index. Especially if index is used many times, as in e.g. t_{dot} from Section 4.3.1, the fact that all these one-hot vectors will be added together means that the reverse derivative of t_{dot} takes $O(n^2)$ time instead of $O(n)$ time! This is unacceptable.

No sparse arrays. The reader may wonder if we can solve this one-hot problem by representing cotangent arrays (i.e. the backpropagated derivatives in *eval*) sparsely. Indeed, by giving array cotangents a sparse runtime representation, creating a one-hot array becomes a constant-time operation, and with a sufficiently clever reduction implementation, an array of such sparse cotangents can even be summed relatively efficiently. However, in practice, the majority of array cotangents are, or become, dense, and it is well-known that performing array operations on a sparse array that is actually completely full (i.e. dense data with a sparse representation) has significant overhead as compared to working with dense arrays directly. Furthermore, sparse arrays do nothing to alleviate the Delta explosion problem, and our solution to the Delta explosion problem also mostly addresses the one-hot problem anyway. Hence, we ignore sparse arrays as a potential solution in this chapter.

Mutable accumulators. Another way the one-hot problem can likely be addressed is to perform a Cayley transform, somewhat similar to the one used in Section 3.5, and replace one-hot vectors by local modifications of a mutable gradi-

ent accumulator. However, this still does not solve the Delta explosion problem, and avoiding mutable updates keeps the algorithm purely functional.

4.3.5 Dual arrays with bulk operations: Our solution

As already remarked at the end of Section 4.3.1, the computation paths that the lambda invocations in a build1 actually take are in practice often extremely similar. In the approach taken in this chapter, we make the most of this observation: we design a code transformation that *eliminates* the higher-order build1 operation and turns it into first-order bulk array operations that can be differentiated neatly as-is. This code transformation "pushes" build1 and index down into expressions, and thus looks a lot like a certain kind of vectorisation, or "unfusion". Representative rules are the following:

$$\begin{aligned} \mathsf{build1}\ k\ (\lambda i.\ op\ t\ u) &\quad\rightsquigarrow\quad op\ (\mathsf{build1}\ k\ (\lambda i.\ t))\ (\mathsf{build1}\ k\ (\lambda i.\ u)) \\ \mathsf{index}\ (\mathbf{let}\ x = v\ \mathbf{in}\ u)\ ix &\quad\rightsquigarrow\quad \mathbf{let}\ x = v\ \mathbf{in}\ \mathsf{index}\ u\ ix \end{aligned}$$

To avoid ascribing even more meanings to the word "vectorisation", we call our transformation the *bulk-operation transformation*, or BOT.

As an example, consider the following source program (fragment):

$$\mathsf{build1}\ k\ (\lambda i.\ \mathsf{index}\ a\ [i] + 1)$$

The BOT will turn this into the following:[21]

$$\mathsf{gather}\ [k]\ a\ (\lambda i.\ i) + \mathsf{replicate}\ k\ 1$$

The crucial point is that these more specific array combinators do not suffer from the Delta explosion problem like build1 does. In fact, their derivatives are quite small; examples, including gather and replicate, are given in Figs. 4.15 and 4.16 in Section 4.5.[22]

The result of the BOT is thus that the user can write explicitly indexed code using 'build' (as well as derived operations such as 'map'), yet the AD algorithm can ignore the existence of 'build' and work solely on effectively differentiable bulk array operations. This solves the Delta explosion problem from Section 4.3.3.

The one-hot problem. The BOT also has an effect on array indexing because an indexing operation inside build1 is turned into a bulk gather operation, like in

[21] Of course, if a has length k, then $\mathsf{gather}\ [k]\ a\ (\lambda i.\ i) = a$, but in general a gather is required.

[22] Elementwise-broadcasted primitive operations have an elementwise forward derivative; operations like gather have a small derivative because the lambda passed to gather need not be differentiated. We will see this again in Section 4.5.3.

the small example above. This is a big improvement: where the reverse derivative of index was a one-hot array, the reverse derivative of gather is a "multi-hot" array that contains non-zero values at all the positions that are read by at least one of the indexing operations collected together in that gather, and zeros elsewhere. In the simple example above, the entirety of a is used (assuming a has length k), so this "multi-hot" derivative of gather is actually fully dense.

More generally, it still holds that one usually does not "ignore" a large fraction of an array — and if one does, there is typically some other part of the program that conversely uses just the part that was ignored here. Hence, we expect that in practice, these "multi-hot" arrays arising from the reverse derivative of 'gather' will be quite dense. For the programs for which this is true, the BOT not only solves the Delta explosion problem, it also solves the indexing one-hot problem.

No general fold. The downside of the BOT is that while build1 is fully supported, other second-order array operations like 'foldl' would significantly complicate the algorithm. The reason is that while we could eliminate the higher-orderness inherent in build1, we cannot eliminate a higher-order fold in the same way, so elementwise code remains in the program to be differentiated, and the Delta explosion problem returns. Thus, such other second-order array operations are *unsupported* in this chapter. However, a general reduction operation (as opposed to the typical first-order ones, such as 'sum' and 'maximum', which are supported just fine) is much less common in typical numerical code than a general elementwise computation, so the algorithm remains useful even with this limitation.

We describe the full transformation in Section 4.4.

4.3.6 Structure of the algorithm exposition

The full description of the algorithm consists of three parts:

- Section 4.4: The bulk-operation transformation that eliminates build1 from the input program.
- Section 4.5: The adaptation of the dual-numbers reverse AD algorithm from Section 4.1 to work on dual arrays. This works out and results in an efficient gradient computation because there is no *active* (roughly: differentiable) non-broadcasted elementwise computation any more.
- Section 4.6: Making the reverse pass (in particular, *eval*) symbolic. This allows us to differentiate a program *once* and run it on many inputs, solving a problem identified in Section 4.1.

4.4 Bulk-operation transformation

As introduced in Section 4.3.5, the aim of the bulk-operation transformation (BOT) is to eliminate 'build1', and as much as possible 'index', from the core language. This allows users to write explicitly indexed code, but lets the AD algorithm of Section 4.5 work on mostly first-order code. As a result, (1) the Delta trace generated by the AD-transformed code will be small (it does not refer to individual scalar operations any more, but only the bulk operations that contain them) and (2) projections from large structures (i.e. index and gather) are batched as much as possible, meaning that we generate very few one-hot/multi-hot cotangent arrays. This addresses the two problems (Delta explosion and one-hot cotangent arrays) that we saw in Sections 4.3.1, 4.3.3 and 4.3.4.

4.4.1 The transformation

The BOT is a set of rewrite rules $u \rightsquigarrow v$ on the core language; the rules can be found in Figs. 4.10 to 4.12. The rules are divided into three categories:

1. Fig. 4.10: Rules that "push down" build1 into the expression, eventually eliminating it when we reach a subexpression that is elementary enough. For example, this is the rule for binary operators *op*:

$$\text{build1}\ k\ (\lambda i.\ op\ t\ u) \quad \rightsquigarrow \quad op\ (\text{build1}\ k\ (\lambda i.\ t))\ (\text{build1}\ k\ (\lambda i.\ u))$$

   We see that whenever we build an array elementwise by combining two computations (t and u) with a binary operator, this is rewritten to building two arrays containing the results of t and u, after which we combine those arrays elementwise.

   When rewriting reaches a leaf expression, e.g. some term t that does not mention the index variable i:

$$\text{build1}\ k\ (\lambda i.\ t) \underset{(i \notin FV(t))}{\rightsquigarrow} \text{replicate}\ k\ t$$

   we eliminate build1.

   The careful reader may note that this figure contains rules for build1 k $(\lambda i.\ t)$ for all term formers t, *except* index. The combination build1-of-index is handled in Fig. 4.12, and will be discussed after we discuss the rules for index itself.

2. Fig. 4.11: Rules that "push down" index into the expression. In many cases, we can eventually cancel the index against a suitable, typically elementwise operation. In cases where we cannot, Fig. 4.11 has a missing rule; these

$$
\begin{array}{rcl}
\mathsf{build1}\ k\ (\lambda i.\ i) & \rightsquigarrow & [0, \ldots, k-1] \\
\mathsf{build1}\ k\ (\lambda i.\ t) & \underset{(i \notin FV(t))}{\rightsquigarrow} & \mathsf{replicate}\ k\ t \\
\mathsf{build1}\ k\ (\lambda i.\ \mathbf{let}\ x = v\ \mathbf{in}\ u) & \rightsquigarrow & \mathbf{let}\ x = \mathsf{build1}\ k\ (\lambda i.\ v) \\
 & & \mathbf{in}\ \mathsf{build1}\ k\ (\lambda i.\ u[\mathsf{index}\ x\ [i]/x]) \\
\mathsf{build1}\ k\ (\lambda i.\ \mathsf{cond}\ b\ u\ v) & \rightsquigarrow & \mathsf{build1}\ k\ (\lambda i.\ \mathsf{index}\ [u, v]\ [\mathsf{cond}\ b\ 0\ 1]) \\
\mathsf{build1}\ k\ (\lambda i.\ op\ t\ u) & \rightsquigarrow & op\ (\mathsf{build1}\ k\ (\lambda i.\ t))\ (\mathsf{build1}\ k\ (\lambda i.\ u)) \\
\mathsf{build1}\ k\ (\lambda i.\ op\ t) & \rightsquigarrow & op\ (\mathsf{build1}\ k\ (\lambda i.\ t)) \\
\mathsf{build1}\ k\ (\lambda i.\ \mathsf{sumOuter}\ t) & \rightsquigarrow & \mathsf{sumOuter}\ (\mathsf{tr}\ (\mathsf{build1}\ k\ (\lambda i.\ t))) \\
\mathsf{build1}\ k\ (\lambda i.\ \mathsf{gather}\ sh\ t\ (\lambda is.\ ix)) & \rightsquigarrow & \mathsf{gather}\ (k ::: sh)\ (\mathsf{build1}\ k\ (\lambda i.\ t))\ (\lambda(i ::: is).\ i ::: ix) \\
\mathsf{build1}\ k\ (\lambda i.\ \mathsf{scatter}\ sh\ t\ (\lambda is.\ ix)) & \rightsquigarrow & \mathsf{scatter}\ (k ::: sh)\ (\mathsf{build1}\ k\ (\lambda i.\ t))\ (\lambda(i ::: is).\ i ::: ix) \\
\mathsf{build1}\ k\ (\lambda i.\ [t_1, \ldots, t_n]) & \rightsquigarrow & \mathsf{tr}\ [\mathsf{build1}\ k\ (\lambda i.\ t_1), \ldots, \mathsf{build1}\ k\ (\lambda i.\ t_n)] \\
\mathsf{build1}\ k\ (\lambda i.\ \mathsf{replicate}\ n\ t) & \rightsquigarrow & \mathsf{tr}\ (\mathsf{replicate}\ n\ (\mathsf{build1}\ k\ (\lambda i.\ t))) \\
\mathsf{build1}\ k\ (\lambda i.\ \mathsf{tr}_{b_0,\ldots,b_n}\ t) & \rightsquigarrow & \mathsf{tr}_{0,b_0+1,b_1+1,\ldots,b_n+1}\ (\mathsf{build1}\ k\ (\lambda i.\ t)) \\
\mathsf{build1}\ k\ (\lambda i.\ \mathsf{reshape}\ sh\ t) & \rightsquigarrow & \mathsf{reshape}\ (k ::: sh)\ (\mathsf{build1}\ k\ (\lambda i.\ t))
\end{array}
$$

Figure 4.10: The rules for the BOT for build1, excluding build1 of index. '*op*' is an elementwise binary or unary arithmetic operator.

$$
\begin{array}{rcl}
\mathsf{index}\ t\ [] & \rightsquigarrow & t \\
\mathsf{index}\ (\mathsf{index}\ t\ [u_1, \ldots, u_m])\ [t_1, \ldots, t_n] & \rightsquigarrow & \mathsf{index}\ t\ [u_1, \ldots, u_m, t_1, \ldots, t_n] \\
\mathsf{index}\ (\textbf{let}\ x = v\ \textbf{in}\ t)\ ix & \rightsquigarrow & \textbf{let}\ x = v\ \textbf{in}\ \mathsf{index}\ t\ ix \\
\mathsf{index}\ (\mathsf{cond}\ b\ u\ v)\ [t_1, \ldots, t_n] & \rightsquigarrow & \textbf{let}\ i_1 = t_1\ \textbf{in} \ldots \textbf{let}\ i_n = t_n \\
 & & \textbf{in}\ \mathsf{cond}\ b\ (\mathsf{index}\ u\ [i_1, \ldots, i_n]) \\
 & & \qquad (\mathsf{index}\ v\ [i_1, \ldots, i_n]) \\
\mathsf{index}\ (op\ t\ u)\ [t_1, \ldots, t_n] & \rightsquigarrow & \textbf{let}\ i_1 = t_1\ \textbf{in} \ldots \textbf{let}\ i_n = t_n \\
 & & \textbf{in}\ op\ (\mathsf{index}\ t\ [i_1, \ldots, i_n]) \\
 & & \qquad (\mathsf{index}\ u\ [i_1, \ldots, i_n]) \\
\mathsf{index}\ (op\ t)\ ix & \rightsquigarrow & op\ (\mathsf{index}\ t\ ix) \\
\mathsf{index}\ (\mathsf{sumOuter}\ t)\ ix & \rightsquigarrow & \mathsf{sumOuter}\ (\mathsf{index}\ (\mathsf{tr}_{1,\ldots,n,0}\ t)\ ix) \\
\mathsf{index}\ [t_1, \ldots, t_k]\ [u_1, \ldots, u_n] & \underset{(n>1)}{\rightsquigarrow} & \textbf{let}\ i_2 = u_2\ \textbf{in} \ldots \textbf{let}\ i_n = u_n \\
 & & \textbf{in}\ \mathsf{index}\ [\mathsf{index}\ t_1\ [i_2, ..., i_n], \ldots, \\
 & & \qquad \mathsf{index}\ t_k\ [i_2, ..., i_n]] \\
 & & \quad [u_1] \\
\mathsf{index}\ (\mathsf{replicate}\ k\ t)\ [u_1, \ldots, u_n] & \underset{(n>0)}{\rightsquigarrow} & \mathsf{index}\ t\ [u_2, \ldots, u_n] \\
\mathsf{index}\ (\mathsf{tr}_{b_0,\ldots,b_k}\ t)\ ix & \rightsquigarrow & \mathsf{index}\ (\mathsf{gather}\ sh\ t \\
 & & \qquad (\lambda[i_{b_0}, \ldots, i_{b_k}].\ [i_0, \ldots, i_k])) \\
 & & \quad ix \\
\mathsf{index}\ (\mathsf{reshape}\ sh\ t)\ ix & \rightsquigarrow & \mathsf{index}\ (\mathsf{gather}\ sh\ t\ (\lambda is. \\
 & & \qquad \mathsf{fromLinearIdx}\ (\mathsf{shape}\ t) \\
 & & \qquad\ (\mathsf{toLinearIdx}\ sh\ is))) \\
 & & \quad ix \\
\mathsf{index}\ (\mathsf{gather}\ (k ::: sh)\ t\ (\lambda(i ::: is).\ ix))\ (u ::: ix') & \underset{(n>0)}{\rightsquigarrow} & \mathsf{index}\ (\mathsf{gather}\ sh\ t\ (\lambda is.\ \textbf{let}\ i = u\ \textbf{in}\ ix))\ ix' \\
\mathsf{index}\ (\mathsf{gather}\ []\ t\ (\lambda[].\ ix))\ ix' & \rightsquigarrow & \mathsf{index}\ (\mathsf{index}\ t\ ix)\ ix'
\end{array}
$$

Figure 4.11: The rules of the BOT for index. 'shape t' is a macro that expands to the (statically-known) shape of its argument term; 'toLinearIdx' and 'fromLinearIdx' are macros that, respectively, flatten a multidimensional index into a linear one and re-nest it into a multidimensional one.

$$
\begin{aligned}
&\texttt{build1}\ k\ (\lambda i.\ \texttt{index}\ x\ ix) && \rightsquigarrow && \texttt{gather}\ (k ::: sh)\ x\ (\lambda[i].\ ix)\\
&\texttt{build1}\ k\ (\lambda i.\ \texttt{index}\ c\ ix) && \rightsquigarrow && \texttt{gather}\ (k ::: sh)\ c\ (\lambda[i].\ ix)\\
&\texttt{build1}\ k\ (\lambda i.\ \texttt{index}\ [t_1, \ldots, t_k]\ [t]) && \rightsquigarrow && \texttt{gather}\ (k ::: sh)\\
&&&&& \quad (\texttt{build1}\ k\ (\lambda i.\ [t_1, \ldots, t_k]))\\
&&&&& \quad (\lambda[i].\ [i, t])\\
&\texttt{build1}\ k\ (\lambda i.\\
&\quad \texttt{index}\ (\texttt{scatter}\ sh\ t\ (\lambda is_2.\ ix_2))\\
&\quad ix) && \overset{(|ix|>0)}{\rightsquigarrow} && \texttt{gather}\ (k ::: sh)\\
&&&&& \quad (\texttt{build1}\ k\ (\lambda i.\\
&&&&& \quad\ \texttt{scatter}\ sh\ t\ (\lambda is_2.\ ix_2)))\\
&&&&& \quad (\lambda[i].\ i ::: ix)
\end{aligned}
$$

Figure 4.12: Rules for the BOT for build1-of-index. Recall that x and c refer to variables and constants, respectively. Note that all index-headed forms that do not appear on the left-hand side here, are rewritten away in Fig. 4.11.

are: index of a variable reference, of a constant array, of combined arrays $[t_1, \ldots, t_n]$, and of scatter. These four forms are precisely the normal forms for rewriting index listed in Theorem 2 below. To ensure that we can still always eliminate build1, the final figure (Fig. 4.12) contains rules that commute build1 below index for these four normal forms.

3. Fig. 4.12: Rules for commuting build1 under index (turning the index into a gather simultaneously). These belong with the list of build1-rules from Fig. 4.10, but are set in a separate figure to make it easier to discuss them separately.

   While these rules preserve semantics and time complexity, the 3rd and 4th rule in this figure do not preserve memory usage under sequential execution. For example, when sequentially executing the left-hand side of the fourth rule:

   $$\texttt{build1}\ k\ (\lambda i.\ \texttt{index}\ (\texttt{scatter}\ sh\ t\ (\lambda is_2.\ ix_2))\ ix)$$

   each scatter is executed independently, and its output (which is immediately mostly discarded by the index) can be deallocated before starting on the next i. The right-hand side of the rule, however, first computes *all* scatters before doing a bulk projection from this big array.

   While unfortunate, the situation is not as bad as it may seem, because when executing the left-hand side in *parallel*, especially on massively parallel hardware like a GPU, many or even all of the scatters would be executed in

parallel. The resulting memory usage is less than the rewritten right-hand side only to the extent that the degree of parallelism is less than k.

Strongly normalising. We can make the behaviour of the rewrite system more formal by looking more precisely at its normal forms. Indeed, the rewrite system is strongly normalising, meaning that rewriting terminates (for all u there is a v, the *normal form* of u, such that $u \rightsquigarrow^* v$ and $\nexists t.\ v \rightsquigarrow t$) and rewriting order is irrelevant (i.e. normal forms are unique). Therefore, one can talk usefully about this set of normal forms (terms in which no more rewrites are possible), and this set tells us something about the capabilities and limitations of the rewrite system.

Theorem 2. *When considering only well-typed terms, the set of normal forms of the rewrite system in Figs. 4.10 to 4.12 consists precisely of those terms t that satisfy the following two properties:*

1. *'build1' does not occur in t.*
2. *Every occurrence of 'index u ix' in t is of the form 'index x $(v ::: ix)$' (for 'x' a variable reference), 'index c $(v ::: ix)$' (for 'c' a constant), 'index $[\ldots]$ $[v]$', or 'index (scatter _ _ _) $(v ::: ix)$'.*

Proof. For (1): assume there is a normal form t that contains build1; then t contains a subterm $t' \coloneqq \text{build1}\ k\ (\lambda i.\ b)$ for some term b where b does *not* contain build1. Thus b is headed by one of the other syntactic forms in Fig. 4.7, and for each of those (note that if a variable x is unequal to i, we certainly have $i \notin FV(x)$) there is a left-hand side in Figs. 4.10 and 4.12 that then matches t'. Therefore t', and thus t, can be rewritten, contradicting normality of t. Hence, there is no such t after all.

For (2): similarly, assume there is a normal form t that contains a subterm $t' \coloneqq \text{index}\ u\ ix$ that does not match any of the stated forms. By (1), u does not contain build1. Then it can be verified using Figs. 4.7 to 4.9 that one of the left-hand sides of Fig. 4.11 matches t', leading to the same contradiction as before, meaning that there is no such t. □

From property (1) of Theorem 2 we know that we have successfully eliminated build1 from the source program by applying the transform. Property (2) is unfortunately more nuanced, because we cannot always fully eliminate index. The upside is that it certainly cannot occur inside build1 any more — because there are no more build1s to occur inside of in the first place. Furthermore, the only other places in the grammar (Fig. 4.7) where a term is executed multiple times are inside the lambda argument to gather and scatter, and because the output type of those lambdas is discrete (namely, an index), their bodies need not be differentiated (see Section 4.5.3), so no one-hots are generated.

Together, this means that the number of one-hots created in the derivative program is at most the number of lexical 'index' occurrences in the source program, which is not too large.

4.4.2 Core language design justification

The BOT-induced restrictions on the core language listed in Section 4.2.1 can be better justified now that we have the BOT rules in front of us.

Static shapes. The type system of the core language (Figs. 4.8 and 4.9) ensures that all array shapes are statically known. This requirement is a weakening of the actual requirement: "the BOT must not get stuck", or more precisely: all the intermediate values computed in a build1-lambda must have shapes that are independent of the index at which the lambda is called.

Let us look at an example to see why this requirement exists. Suppose that the core language contained a primitive, called 'filter', of which the output shape is unknown statically:

$$\frac{\Gamma, x : \rho \vdash s : \text{Array } [] \text{ Bool} \qquad \Gamma \vdash t : \text{Array } sh\ \rho}{\Gamma \vdash \text{filter } (\lambda x.\ s)\ t : \text{Array } ??\ \rho}$$

The semantics is to filter an array on a predicate: $\text{filter } (\lambda x.\ x > 4)\ [3, 8, -16, 7, 2] = [8, 7]$. Of course, there is no sensible shape to substitute for '??' here — which is the point — but suppose that we had a weaker type system that allowed this.

The question now is: what does $\text{build1 } 10\ (\lambda i.\ \text{filter } (\lambda x.\ s)\ t)$ vectorise to?

Regardless of what term this would map to, the array that it ought to produce is *not rectangular*: it is not a regular multi-dimensional array, also called a *jagged array*. Such arrays pose problems with efficient indexing, bounds checking of indexing, semantics of array transposition, etc. Hence, we disallow such arrays: all our arrays are regular. This implies that the computation in a 'build1' lambda, *including all its intermediate values*, must have uniform shapes over all values of the index variable i. This is, strictly speaking, a weaker requirement than static shapes, but it is not very much weaker in practice, and static shapes are much easier to enforce for us and to understand for a user.

Strict conditionals. Consider the BOT rule for build1-of-cond:

$$\text{build1 } k\ (\lambda i.\ \text{cond } b\ u\ v) \quad \rightsquigarrow \quad \text{build1 } k\ (\lambda i.\ \text{index } [u, v]\ [\text{cond } b\ 0\ 1])$$

The reason the conditionals in our language are strict is that this justifies the right-hand side of this translation: it computes the two arguments first, then

picks the correct one using index. Having proper conditionals would not compose nearly as well with the BOT as these strict conditionals.

To nevertheless support some algorithms that would otherwise require proper conditionals, it is important that our built-in operations never crash: this permits the user, at least when reasoning semantically, to think of cond as a proper conditional. For example, consider the following program that concatenates an array a (assumed in scope with length 10) to itself:

$$\text{build1 } 20\ (\lambda i.\ \text{cond } (i < 10)\ (\text{index } a\ [i])\ (\text{index } a\ [i - 10]))$$

After the BOT, this code will evaluate both index expressions for the *full* domain $\{0, \ldots, 19\}$ instead of only their intended domain. Hence, this code only works because our index operation, as well as other normally partial operations (such as gather and the division operator), check their arguments and still return a value even if the arguments are invalid.

Single expression. The core language does not admit separate top-level functions: the program must be a single expression with let-bindings. The BOT requires this because the generated code for any particular subterm depends on the context in which it runs, all the way to the top level of the program, so all of this context must be *visible* to the code transformation. Modularity via e.g. top-level functions would make this impossible.

This limitation can be ameliorated somewhat by an inlining pass before the BOT and the AD algorithm proper runs, that eliminates any user-written top-level functions by simply inlining them at every call site. This may blow up the program significantly in some cases, but note that the trace that AD will generate is on the order of the size of the *fully inlined* program anyway.

4.5 Dual arrays: Differentiating bulk array programs

In the existing scalar-level dual-numbers reverse AD algorithm (described in Section 4.1), each scalar is considered an independent object during differentiation. We saw in Section 4.3 that while this approach can be easily and naturally extended to arrays, it results in very slow gradient code. As a solution to this problem, we lifted the granularity of the algorithm to entire arrays of scalars (thus creating *dual arrays* instead of "dual numbers"); the idea here is that a single array operation translates to very many individual scalar operations, and the fewer operations in the program to be differentiated, the lower the overhead introduced by differentiation.

To allow the user to write code that nevertheless works on individual scalars (using build1 and derived operations such as 'map'), the BOT from the previous

$$D[\text{Array } sh\, \mathbb{R}] = (\text{Array } sh\, \mathbb{R}, \text{Delta } sh)$$
$$D[\text{Array } sh\, \text{Int}] = \text{Array } sh\, \text{Int}$$
$$D[\text{Array } sh\, \text{Bool}] = \text{Array } sh\, \text{Bool}$$

data Delta *sh* **where**
  — scalar linear maps and input
  Zero :: Delta *sh*
  Input :: DVarName → Delta *sh*
  Add :: Delta *sh* → Delta *sh* → Delta *sh*
  Scale :: Array *sh* $\mathbb{R}$ → Delta *sh* → Delta *sh*
  — encoding sharing
  Share :: ID → Delta *sh* → Delta *sh*
  — linear array operations
  Index :: Delta $[k_1, \ldots, k_n]$ → Ix m → Delta $[k_{m+1}, \ldots, k_n]$
  SumOuter :: Delta $(k ::: sh)$ → Delta *sh*
  Gather :: Delta $[k_1, \ldots, k_{m_2}, k_{m_2+1}, \ldots, k_n]$ → (Ix m_1 → Ix m_2)
    → Delta $[k'_1, \ldots, k'_{m_1}, k_{m_2+1}, \ldots, k_n]$
  Scatter :: Delta $[k_1, \ldots, k_{m_1}, k_{m_1+1}, \ldots, k_n]$ → (Ix m_1 → Ix m_2)
    → Delta $[k'_1, \ldots, k'_{m_2}, k_{m_1+1}, \ldots, k_n]$
  LitArray :: Array $[k]$ (Delta *sh*) → Delta $(k ::: sh)$
  Replicate :: Delta *sh* → Delta $(k ::: sh)$
  Transpose$_{j_1,\ldots,j_m}$ :: Delta $[k_1, \ldots, k_n]$ → Delta $[k_{j_1+1}, \ldots, k_{j_m+1}, k_{m+1}, \ldots, k_n]$
  Reshape :: Delta *sh* → Delta *sh*′

Figure 4.13: Types for array-level dual-numbers reverse AD. Delta *sh* represents the derivative of a term of type Array *sh* $\mathbb{R}$. Slightly modified in Eq. (4.4) (page 166).

section rewrites build1 into bulk array operations with a bulk derivative.

In this section, we start from the output of the BOT, and explain the dual arrays AD algorithm that we apply to it. Afterwards, in Section 4.6, we will lift the evaluator (the reverse pass) to symbolic tensors to make it possible to differentiate a term once and then compute many different gradients with it.

4.5.1 Types of the transformation

Recall from Section 4.2.1 that the type system of the core language is very simple:

$$\rho ::= \mathbb{R} \mid \text{Int} \mid \text{Bool}$$
$$\sigma, \tau ::= \text{Array } sh\, \rho$$

Because of this, the type transformation of the AD algorithm is also very simple (Fig. 4.13, top).

As witnessed by $D[\text{Array } sh\ \mathbb{R}] = (\text{Array } sh\ \mathbb{R}, \text{Delta } sh)$, the output of this code transformation uses tuples where the input did not. This is because the algorithm is, in a way, still *dual* numbers reverse AD. We elide the precise grammar and type system extensions to the core language that allow tuples at the top level (i.e. not as elements of arrays!); there are no surprises here.

While the type transformation is simple, we do need to add a number of constructors to the Delta data type; we already observed this in Section 4.3. We can still use Add and Scale for the binary and unary arithmetic operators in the language — where the scaling constant in Scale becomes array-valued, and the scaling is performed elementwise — which is why they still appear in Delta in Fig. 4.13. This way, we avoid a proliferation of Delta constructors, one for each broadcasted arithmetic operator. For the other array operations, however, we generally have a bespoke Delta constructor whose semantics is precisely its forward derivative.

For indices into multidimensional arrays, we use the Ix data type from Section 4.3.4:

$$\begin{aligned}&\textbf{data}\ \text{Ix}\ k\ \textbf{where}\\ &\quad \text{IZ} \quad :: \text{Ix}\ 0\\ &\quad (:::) :: \text{Int} \to \text{Ix}\ k \to \text{Ix}\ (k+1)\end{aligned}$$

As an example of how these Delta terms correspond to forward derivatives, consider Index. This Delta constructor represents the forward derivative of the core primitive 'index', which has the following typing rule:

$$\frac{\Gamma \vdash t : \text{Array}\ [k_1, \ldots, k_n]\ \rho \qquad \Gamma \vdash ix \text{ is an } m\text{-dim. index} \qquad m \leq n}{\Gamma \vdash \text{index}\ t\ ix : \text{Array}\ [k_{m+1}, \ldots, k_n]\ \rho}$$

Because the result of 'index t ix' is simply the ix'th element (more accurately, $(n-m)$-dimensional subarray) of t, the forward derivative of index t ix is also simply the ix'th element (subarray) of the forward derivative of t. Delta terms are programs that *compute* the forward derivative, so given the Delta term computing the forward derivative of t, Index should index that result at this same position ix. This is precisely the semantics that we give to the Index constructor.

Note that it is unsurprising that the Delta term Index does precisely the same thing as index in the core language; as observed in Section 4.3, this follows from the fact that array indexing is an algebraically linear operation.[23] In fact, all our array operations, except for the broadcasted arithmetic operations, are algebraically linear, thus all Delta constructors apart from the basic ones necessary to differentiate primitive arithmetic operations (Zero, Input, Add, Scale and Share) mirror the semantics of their corresponding core language operation.

[23]The (forward) derivative is the best linear approximation of a function, so if a function is already linear, then its best linear approximation is itself.

$$x_1 : \tau_1, \ldots, x_n : \tau_n, \Gamma \vdash t : \tau \quad \rightsquigarrow \quad x_1 : D[\tau_1], \ldots, x_n : D[\tau_n], \Gamma \vdash D'_\Gamma[t] : \tau$$

$$\begin{aligned}
D'_\Gamma[x] = {} & x \quad (\text{if } x \in \Gamma) && \text{— Internal local variable} \\
& \text{fst } x \quad (\text{else, if } x :: \text{Array } sh\, \mathbb{R}) && \text{— Free dual-number variable} \\
& x \quad (\text{otherwise}) && \text{— Free discrete variable} \\
D'_\Gamma[c] &= c \\
D'_\Gamma[\textbf{let } x = u \textbf{ in } v] &= \textbf{let } x = D'_\Gamma[u] \textbf{ in } D'_{\Gamma,x}[v] \\
D'_\Gamma[\text{cond } t\ u\ v] &= \text{cond } D'_\Gamma[t]\ D'_\Gamma[u]\ D'_\Gamma[v] \\
D'_\Gamma[op\ u\ v] &= op\ D'_\Gamma[u]\ D'_\Gamma[v] \\
D'_\Gamma[op\ t] &= op\ D'_\Gamma[t] \\
D'_\Gamma[\text{index } t\ [t_1, \ldots, t_n]] &= \text{index } D'_\Gamma[t]\ [D'_\Gamma[t_1], \ldots, D'_\Gamma[t_n]] \\
D'_\Gamma[\text{sumOuter } t] &= \text{sumOuter } D'_\Gamma[t] \\
D'_\Gamma[\text{gather } sh\ t\ (\lambda is.\ [t_1, ..., t_n])] &= \text{gather } sh\ D'_\Gamma[t]\ (\lambda is.\ [D'_{\Gamma,is}[t_1], ..., D'_{\Gamma,is}[t_n]]) \\
D'_\Gamma[\text{scatter } sh\ t\ (\lambda is.\ [t_1, ..., t_n])] &= \text{scatter } sh\ D'_\Gamma[t]\ (\lambda is.\ [D'_{\Gamma,is}[t_1], ..., D'_{\Gamma,is}[t_n]]) \\
D'_\Gamma[[t_1, \ldots, t_n]] &= [D'_\Gamma[t_1], \ldots, D'_\Gamma[t_n]] \\
D'_\Gamma[\text{replicate } k\ t] &= \text{replicate } k\ D'_\Gamma[t] \\
D'_\Gamma[\text{tr}_{k_1,\ldots,k_n}\ t] &= \text{tr}_{k_1,\ldots,k_n}\ D'_\Gamma[t] \\
D'_\Gamma[\text{reshape } sh\ t] &= \text{reshape } sh\ D'_\Gamma[t] \\
D'_\Gamma[\text{build1 } k\ (\lambda i.\ t)] &= \text{build1 } k\ (\lambda i.\ D'_{\Gamma,i}[t])
\end{aligned}$$

Figure 4.14: The non-differentiating code transformation of Section 4.5.2.

4.5.2 A non-differentiating transformation

Before we can move on to the term transformation, we have to introduce yet another code transformation ($D'[-]$) that makes a term compatible with surrounding differentiated code, but does not actually differentiate it. Not differentiating the term in question means that the transformed code need not live inside the monad. We will use this non-differentiating transformation on the index functions passed to gather and scatter in the actual differentiating transformation in Section 4.5.3.

The rules can be found in Fig. 4.14. Note that in contrast to the *differentiating* transformation $D[-]$, the type of the transformed expression is not IdGen $D[\tau]$ but instead simply τ: only the types of the free variables change.

Because only the *free* variables change type, $D'[-]$ needs to take special care to distinguish free variables (bound outside the term initially passed to $D'[-]$) from local variables (those bound inside). In Fig. 4.14, $x_1, \ldots, x_n$ are the free variables and Γ contains the locally-bound variables. The transformation is

indexed[24] by Γ so that the case for variable references x, where the actual logic happens, can choose the correct result term depending on whether x is locally bound (and hence not differentiated, so always of the original type) or free (and hence transformed to a dual number if the original x was of type Array $sh\ \mathbb{R}$).

4.5.3 The term transformation

We now have all the pieces to extend the (differentiating) code transformation from Section 4.1 to the core array language from Section 4.2.1 — or more precisely, the fragment of the core language that is produced by the BOT (Section 4.4): all but build1.

The term transformation is given in Figs. 4.15 and 4.16.[25] The extension generally follows the pattern set out in Section 4.1, but some aspects benefit from closer examination.

- In the core language, we have operations that apply both[26] to the *dualised type* (Array $sh\ \mathbb{R}$) and to non-dualised types (arrays of non-$\mathbb{R}$ elements), and yet they look at the internal structure of their argument. In the simple language in Section 4.1, we did not have such constructs: a construct either monomorphically worked on scalars (e.g. $(\times_{\mathbb{R}})$, which got a derivative that works on dual numbers specifically) or kept the values of the scalars as-is (e.g. 'fst', pairing or lambda-abstraction, each of which got derivatives oblivious of the existence of dual numbers). The presence of operations in the core language that mix the two (e.g. gather, replicate) means that their derivative under $D[-]$ necessarily differs depending on whether they work on an array of scalars or not.

- Despite the fact that e.g. the terms $t_1, \ldots, t_n$ in '$\mathsf{index}\ t\ [t_1, \ldots, t_n]$' are of type Array sh Int and that $D[\text{Array } sh \text{ Int}] = \text{Array } sh \text{ Int}$, we cannot simply use those $t_1, \ldots, t_n$ in the differentiated program as-is: the types of their free variables are wrong. While using $D[t_i]$ instead would do the trick, we use $D'[t_i]$ to avoid potentially building Delta terms that will only be discarded later. In the non-scalar version of $D[\mathsf{index}]$, even the array being indexed is not dualised, so we can convert that term using $D'[-]$ too. ('ε' denotes the empty environment.)

[24] For conciseness, we elide types in the bindings added to Γ in the rules for **let**, gather, scatter and build1.

[25] Note the lack of expressivity of our core language syntax here, as noted in Section 4.2.1: because index lists are not first-class in the language, the expression under the lambda in a gather is a list of terms, not a term that produces a list. We present the core language this way for simplicity only.

[26] They are "polymorphic", albeit not by explicit polymorphism in the language, but instead by a custom typing rule.

$$
\begin{aligned}
D[c] &= \textbf{return } (c, \text{Zero}) \\
D[\text{cond } t\ u\ v] &= \textbf{do } x \leftarrow D[t] \\
&\quad\ y \leftarrow D[u] \\
&\quad\ z \leftarrow D[v] \\
&\quad\ \textbf{return } (\text{cond } x\ y\ z)
\end{aligned}
\qquad\qquad
\begin{aligned}
D[x] &= \textbf{return } x \\
D[\textbf{let } x = u \textbf{ in } v] &= \textbf{do } x \leftarrow D[u] \\
&\quad\ D[v]
\end{aligned}
$$

$$
\begin{aligned}
D[\text{index } t\ [t_1, \ldots, t_n]] = {} & \text{— } \textit{If } t :: \text{Array } sh\ \mathbb{R}\textit{:} \\
& \textbf{do } (x, d) \leftarrow D[t] \\
& \quad \textbf{let } i_1 = D'_\varepsilon[t_1]; \ldots; i_n = D'_\varepsilon[t_n] \\
& \quad id \leftarrow \text{genID} \\
& \quad \textbf{return } (\text{index } x\ [i_1, \ldots, i_n] \\
& \qquad\qquad , \text{Share } id\ (\text{Index } d\ [i_1, \ldots, i_n])) \\
& \text{— } \textit{Otherwise (i.e. } t :: \text{Array } sh\ \text{Int } \textit{or } t :: \text{Array } sh\ \text{Bool}\textit{):} \\
& \textbf{return } (\text{index } D'_\varepsilon[t]\ [D'_\varepsilon[t_1], \ldots, D'_\varepsilon[t_n]]) \\
D[\text{sumOuter } t] = {} & \text{— } \textit{If } t :: \text{Array } sh\ \mathbb{R}\textit{:} \\
& \textbf{do } (x, d) \leftarrow D[t] \\
& \quad id \leftarrow \text{genID} \\
& \quad \textbf{return } (\text{sumOuter } x, \text{Share } id\ (\text{SumOuter } d)) \\
& \text{— } \textit{Otherwise:} \\
& \textbf{return } (\text{sumOuter } x) \\
D[\text{gather } sh\ t\ (\lambda is.\ [t_1, \ldots, t_n])] = {} & \text{— } \textit{If } t :: \text{Array } sh\ \mathbb{R}\textit{:} \\
& \textbf{do } (x, d) \leftarrow D[t] \\
& \quad id \leftarrow \text{genID} \\
& \quad \textbf{return } (\text{gather } sh\ x\ (\lambda is.\ [D'_{is}[t_1], \ldots, D'_{is}[t_n]]) \\
& \qquad\qquad , \text{Share } id\ (\text{Gather } d\ (\lambda is.\ [D'_{is}[t_1], \ldots, D'_{is}[t_n]]))) \\
& \text{— } \textit{Otherwise:} \\
& \textbf{return } (\text{gather } sh\ D'_\varepsilon[t]\ (\lambda is.\ [D'_{is}[t_1], \ldots, D'_{is}[t_n]])) \\
D[\text{scatter } sh\ t\ (\lambda is.\ [t_1, \ldots, t_n])] = {} & \text{— } \textit{Elided to save space; analogous to } \textit{gather}\textit{.} \\
D[[t_1, \ldots, t_n]] = {} & \text{— } \textit{If } t_i :: \text{Array } sh\ \mathbb{R}\textit{:} \\
& \textbf{do } (x_1, d_1) \leftarrow D[t_1]; \ldots; (x_n, d_n) \leftarrow D[t_n] \\
& \quad id \leftarrow \text{genID} \\
& \quad \textbf{return } ([x_1, \ldots, x_n], \text{Share } id\ (\text{LitArray } [d_1, \ldots, d_n])) \\
& \text{— } \textit{Otherwise:} \\
& \textbf{return } [D'_\varepsilon[t_1], \ldots, D'_\varepsilon[t_n]] \\
D[\text{replicate } k\ t] = {} & \text{— } \textit{If } t :: \text{Array } sh\ \mathbb{R}\textit{:} \\
& \textbf{do } (x, d) \leftarrow D[t] \\
& \quad id \leftarrow \text{genID} \\
& \quad \textbf{return } (\text{replicate } k\ x, \text{Share } id\ (\text{Replicate } d)) \\
& \text{— } \textit{Otherwise:} \\
& \textbf{return } (\text{replicate } k\ D'_\varepsilon[t])
\end{aligned}
$$

Figure 4.15: The AD transformation on the core array language. Continued in Fig. 4.16.

$$
\begin{aligned}
D[\mathsf{tr}_{k_1,\ldots,k_n}\ t] = {} & \text{— } \textit{If } t :: \text{Array } sh\ \mathbb{R}\textit{:} \\
& \textbf{do}\ (x, d) \leftarrow D[t] \\
& \quad id \leftarrow \text{genID} \\
& \quad \textbf{return}\ (\mathsf{tr}_{k_1,\ldots,k_n}\ x, \text{Share } id\ (\text{Transpose}_{k_1,\ldots,k_n}\ d)) \\
& \text{— } \textit{Otherwise:} \\
& \textbf{return}\ (\mathsf{tr}_{k_1,\ldots,k_n}\ D'_\varepsilon[t]) \\
D[\mathsf{reshape}\ sh\ t] = {} & \text{— } \textit{If } t :: \text{Array } sh\ \mathbb{R}\textit{:} \\
& \textbf{do}\ (x, d) \leftarrow D[t] \\
& \quad id \leftarrow \text{genID} \\
& \quad \textbf{return}\ (\mathsf{reshape}\ sh\ x, \text{Share } id\ (\text{Reshape } sh\ d)) \\
& \text{— } \textit{Otherwise:} \\
& \textbf{return}\ (\mathsf{reshape}\ sh\ D'_\varepsilon[t]) \\
D[t_1 \times_{\text{Array } sh\ \mathbb{R}} t_2] = {} & \textbf{do}\ (x_1, d_1) \leftarrow D[t_1]; (x_2, d_2) \leftarrow D[t_2] \\
& \quad id \leftarrow \text{genID} \\
& \quad \textbf{return}\ (x_1 \times_{\text{Array } sh\ \mathbb{R}} x_2 \\
& \qquad\qquad , \text{Share } id\ (\text{Add } (\text{Scale } x_2\ d_1)\ (\text{Scale } x_1\ d_2))) \\
D[t_1 \times_{\text{Array } sh\ \text{Int}} t_2] = {} & \textbf{return}\ (D'_\varepsilon[t_1] \times_{\text{Array } sh\ \text{Int}} D'_\varepsilon[t_2]) \\
& \text{— } \textit{etc. other broadcasted arithmetic operations on arrays}
\end{aligned}
$$

Figure 4.16: Continuation of Fig. 4.15.

The same holds for the terms $t_1, \ldots, t_n$ in '$\mathsf{gather}\ sh\ t\ (\lambda is.\ [t_1, \ldots, t_n])$', but there the usage of $D'[t_i]$ is essential: $D[t_i]$ would run in the IdGen monad, and gather and Gather expect a non-monadic function.

- The reader might wonder: if avoiding elementwise scalar computation in build1 is what we did the whole BOT for, why is the elementwise computation in gather fine (despite the fact that it may indeed contain scalar computation too, if the results are subsequently converted back to integers!)? The answer is the same as for why we could use $D'[-]$ for those terms: any scalar computation that happens inside the function passed to gather cannot *continuously* influence the final program result (because it can only influence said result through the discrete, integral results of that function), so it does not need to be differentiated. Hence this computation does not end up as sub-traces in Delta, and the Delta explosion problem of Section 4.3.1 does not arise.

- The arithmetic operations in Fig. 4.16 generalise straight from Fig. 4.3 in Section 4.1, with operations on Array $sh\,\mathbb{R}$ getting differentiated and operations on arrays with discrete elements being fully preserved.

- Finally, the reader might wonder if, at least for dualised types (i.e. Array $sh\,\mathbb{R}$), the derivative for conditionals could be written instead as follows:

$$
\begin{aligned}
D[\mathsf{cond}\ t\ u\ v] = \mathbf{do}\ \mathbf{let}\ & x = D'_\varepsilon[t] \\
& (y, d_1) \leftarrow D[u] \\
& (z, d_2) \leftarrow D[v] \\
& id \leftarrow \mathrm{genID} \\
& \mathbf{return}\ (\mathsf{cond}\ x\ y\ z, \mathsf{Share}\ id\ (\mathsf{Cond}\ x\ d_1\ d_2))
\end{aligned}
$$

with a new Delta constructor Cond storing a boolean and two Delta terms. The idea of this change would be to allow proper, lazy conditionals in the input language to this part of the AD algorithm, even if the BOT still does not support them.

Unfortunately, this does not work. Extending the regular evaluator (Fig. 4.18 below) to Cond would work fine, but if we stop there, we have won nothing: we have just created a bigger Delta term of which we interpret only half. The big advantage would come only when we make Delta evaluation symbolic in Section 4.6 (Fig. 4.20), but there Cond would suddenly present a problem: it is not statically known which Delta subterm *eval* should descend into, so it is unknown, statically, what contributions should be made to the evaluation state. For this reason, the rule for cond is presented simply as it is in Fig. 4.15.

```
data DMap f g
DMap.empty :: DMap f g        — shorthand: '{}'
DMap.insert :: GCompare f ⇒ f a → g a → DMap f g → DMap f g
DMap.lookup :: GCompare f ⇒ f a → DMap f g → Maybe (g a)
(DMap.!) :: GCompare f ⇒ f a → DMap f g → g a
DMap.delete :: GCompare f ⇒ f a → DMap f g → DMap f g
DMap.insertWith :: GCompare f ⇒ (g a → g a → g a)
                                → f a → g a → DMap f g → DMap f g
DMap.maxViewWithKey :: DMap f g → Maybe (∃a. (f a, g a), DMap f g)
```

Figure 4.17: Types of methods on DMap, as provided by the package `https://hackage.haskell.org/package/dependent-map`. (The existential in the type of 'DMap.maxViewWithKey' is encoded as a newtype in Haskell.) The 'DMap.!' method is a partial version of DMap.lookup.

4.5.4 The evaluator

In Section 4.1 and in [Krawiec et al. 2022, Fig. 11], the reverse pass (evaluation of Delta terms) works on a state represented by Map values keyed by DVarName and ID. This was admissible because every input, as well as every node ID, corresponded to a single scalar, and hence the maps were homogeneous. Now that DVarName and ID correspond to entire arrays of scalars (which may differ in shape and hence in type), the evaluation state needs to contain *heterogeneous* maps. To retain type safety, we use *dependent maps*: if normal maps (associative arrays) can be seen as a collection of pairs, then a dependent map is a collection of *dependent* pairs. The types of the methods on a dependent map, as far as we use them, are given in Fig. 4.17.

The point of a dependent map is to be able to map a type-indexed key to a type-indexed value. Our values (arrays and Delta terms) are shape-indexed and will, at least in the reverse pass, always contain scalars, hence shape-indexing is sufficient. Previously, in the reverse pass for the scalar-level algorithm in Section 4.1 (Fig. 4.5), the evaluation state looked as follows:

```
data ES = ES
  { grad :: Map DVarName ℝ    — input cotangents: will collect final gradient
  , dfrag :: Map ID Delta     — delta fragments
  , accum :: Map ID ℝ }       — accumulated node cotangents
```

We see that our map keys are DVarName (for input values) and ID (for intermediate nodes in the graph). These types gain a type parameter indicating the shape

of the array they refer to:

```
data Delta sh where
  Input :: DVarName sh → Delta sh
  Share :: ID sh → Delta sh → Delta sh
  — ... other constructors ...
```
(4.4)

And the evaluation state changes accordingly:[27]

```
data ES = ES
  { grad :: DMap DVarName (λsh. Array sh ℝ)
  , dfrag :: DMap ID Delta
  , accum :: DMap ID (λsh. Array sh ℝ) }
```

Note our use of a type-level lambda here; in Haskell, this needs to be encoded using a newtype.

With this updated typing, we can update the reverse pass for arrays. The $D[-]$ code transformation gave the *forward derivatives* of our language constructs, expressed in the language of Delta terms. The reverse pass now has the task of transposing these forward derivatives to *reverse derivatives*. First, we change the type of *reversePass* to work with arrays instead of scalars (parts changed from Fig. 4.5 highlighted in red):

```
reversePass :: Array sh ℝ → Delta sh → DMap DVarName (λsh. Array sh ℝ)
reversePass c d = grad (backprop (eval c d (ES {} {} {})))
```

taking an array-valued incoming cotangent to be immediately passed to *eval*.

The definition of *backprop* from Fig. 4.5 remains unchanged except for a simple textual substitution of "Map" to "DMap". The major change is in *eval*, which now has to (1) deal with array values, and (2) handle more Delta constructors than before. The result is shown in Fig. 4.18; let us discuss some important aspects.

When comparing Fig. 4.18 with Fig. 4.5, we see that the incoming cotangent c is now array-valued, just like the cotangent accumulators in the evaluation state and the scaling constant in Scale. Hence, the addition and scaling operators from $eval_4$ in Fig. 4.5 have to be broadcasted (i.e. evaluated elementwise) on arrays now; this we indicate with $\vec{+}$ and $\vec{\cdot}$.

[27] For operations on these DMaps to typecheck, ID and DVarName must implement the `GCompare` type class. They can do so if they, in addition to the integer ID/name itself, also contain a singleton representing the type index (i.e. the rank) on the value level. Key equality checking will then also compare the singletons for equality. Because a list of `GHC.TypeLits.SNats` (themselves simply integers) can suffice for these singletons, this need not be very expensive, but if the overhead is still unacceptable, the only thing that removing the singleton (and using an unsafe coercion to conjure up the type equality evidence) compromises is *confidence* in the algorithm's type-correctness — if two IDs have equal identifiers but different types, that is a bug in our code.

$$
\begin{aligned}
&\mathit{eval} :: \text{Array } \mathit{sh}\,\mathbb{R} \to \text{Delta } \mathit{sh} \to \text{ES} \to \text{ES}\\
&\mathit{eval}\ c\ \text{Zero} &&= \text{id}\\
&\mathit{eval}\ c\ (\text{Input } v) &&= \lambda s.\ s\ \{\ \text{grad} = \text{DMap.insertWith}\ (\vec{+})\ v\ c\ (\text{grad}\ s)\ \}\\
&\mathit{eval}\ c\ (\text{Add } d_1\ d_2) &&= \mathit{eval}\ c\ d_2 \circ \mathit{eval}\ c\ d_1\\
&\mathit{eval}\ c\ (\text{Scale } \mathit{arr}\ d) &&= \mathit{eval}\ (c \vec{\cdot} \mathit{arr})\ d\\
&\mathit{eval}\ c\ (\text{Share } i\ d) &&= \lambda s.\ s\ \{\ \text{dfrag} = \text{DMap.insert}\ i\ d\ (\text{dfrag}\ s)\\
&&&\qquad ,\ \text{accum} = \text{DMap.insertWith}\ (\vec{+})\ i\ c\ (\text{accum}\ s)\ \}\\
&\mathit{eval}\ c\ (\text{Index } d\ i) &&= \mathit{eval}\ (\text{oneHot}\ (\text{shapeDelta}\ d)\ i\ c)\\
&\mathit{eval}\ c\ (\text{SumOuter } d) &&= \mathit{eval}\ (\text{replicate}\ (\text{head}\ (\text{shapeDelta}\ d))\ c)\ d\\
&\mathit{eval}\ c\ (\text{Gather } d\ f) &&= \mathit{eval}\ (\text{scatter}\ (\text{shapeDelta}\ d)\ c\ f)\ d\\
&\mathit{eval}\ c\ (\text{Scatter } d\ f) &&= \mathit{eval}\ (\text{gather}\ (\text{shapeDelta}\ d)\ c\ f)\ d\\
&\mathit{eval}\ c\ (\text{LitArray } \mathit{ds}) &&= \textbf{let}\ [n] = \text{shape}\ \mathit{ds}\\
&&&\quad \textbf{in}\ \mathit{eval}\ (\text{index}\ c\ [n-1])\ (\text{index}\ \mathit{ds}\ [n-1])\\
&&&\qquad \circ \cdots \circ \mathit{eval}\ (\text{index}\ c\ [0])\ (\text{index}\ \mathit{ds}\ [0])\\
&\mathit{eval}\ c\ (\text{Replicate } d) &&= \mathit{eval}\ (\text{sumOuter}\ c)\ d\\
&\mathit{eval}\ c\ (\text{Transpose}_{j_1,\ldots,j_m}\ d) &&= \mathit{eval}\ (\text{tr}_{\text{inversePermutation}(j_1,\ldots,j_m)}\ c)\ d\\
&\mathit{eval}\ c\ (\text{Reshape } d) &&= \mathit{eval}\ (\text{reshape}\ (\text{shapeDelta}\ d)\ c)\ d
\end{aligned}
$$

Figure 4.18: The evaluator in the reverse pass for the array AD algorithm.

In the cases for the array operations, we see that *eval* computes the linear transpose of the forward derivatives described by the Delta term, and applies them to the c argument. The transpose of SumOuter (summation, matrix $\begin{pmatrix}1 & \cdots & 1\end{pmatrix}$) is replication (matrix $\begin{pmatrix}1 & \cdots & 1\end{pmatrix}^\top$); the transpose of a gather is a scatter. This was in fact already true of the existing Delta constructors, but it was somewhat less obvious: the transpose of addition (matrix $\begin{pmatrix}1 & 1\end{pmatrix}$) is duplication (matrix $\begin{pmatrix}1 & 1\end{pmatrix}^\top$), and the transpose of scaling (matrix $\begin{pmatrix}r\end{pmatrix}$) is scaling (matrix $\begin{pmatrix}r\end{pmatrix}^\top = \begin{pmatrix}r\end{pmatrix}$).

In order to compute the transposes of many of the array operations, we need the shape of the original argument arrays to the operation (conveniently equal to the shape of their forward derivative, as computed by the Delta term arguments). For example, in *eval* c (SumOuter d), we need the shape of the array computed by d in order to replicate up the incoming cotangent to a cotangent appropriate for reverse-evaluating d. This shape can be computed by inspection of the Delta term; we implement this in a function 'shapeDelta', the implementation of which we elide. This 'shapeDelta' function can be made constant-time by caching well-chosen explicit shape vectors in Delta constructors.

Finally, we use a few array operations that are not strictly speaking in the core language. The shape function was already used in Fig. 4.11, and returns the shape of its argument array. The oneHot operation produces an array with

one specified entry having a particular value, and zeros elsewhere; it could be implemented as follows:

$$\mathsf{oneHot}\ sh\ idx\ x = \mathsf{gather}\ sh\ [0, x]\ (\lambda idx'.\ \mathsf{cond}\ (idx = idx')\ [1]\ [0])$$

but likely benefits from a specialised implementation.

4.5.5 Wrapper

As before in Section 4.1, this reverse AD algorithm for array programs needs a wrapper to be useful. Compared to the wrapper described in Section 4.1.7, the one for the array AD algorithm is mostly identical; the only wrinkle is that we cannot simplify by assuming the program to differentiate has only one argument (because the core language does not support tuples). The type of the wrapper thus becomes the following:

$$\begin{aligned}
&\text{— } \textit{First argument has free variables: } x_1 : \mathrm{Array}\ sh_1\ \rho_1, \ldots, x_n : \mathrm{Array}\ sh_n\ \rho_n \\
&\mathrm{wrapper} :: \mathrm{AST}\ (\mathrm{Array}\ []\ \mathbb{R}) \\
&\qquad \to (\mathrm{Array}\ sh_1\ \rho_1, \ldots, \mathrm{Array}\ sh_n\ \rho_n) \\
&\qquad \to \mathrm{Array}\ []\ \mathbb{R} \\
&\qquad \to (\mathrm{Array}\ sh_1\ \rho_1, \ldots, \mathrm{Array}\ sh_n\ \rho_n)
\end{aligned}$$

The implementation is completely analogous to the one in Section 4.1.7.

4.6 Compile-time differentiation

This wrapper obtained in Section 4.5.5 takes a term, but returns a *numeric* gradient. In particular, it does this by passing the input to the transformed version of the source program, and *interpreting* (with *reversePass*) the resulting Delta term. Typically, one requires a gradient at many different input points, and with the current setup, this results in interpretation overhead from *reversePass* for every such input point. In this section, we improve upon this by extracting the Delta term from the *unevaluated* transformed program, and evaluating (transposing) this term symbolically.

We first show an example (Section 4.6.1) that illustrates the pipeline so far, and introduces the observation that allows us to symbolically evaluate the Delta term in a forward-differentiated program without having to provide an input first. Then, in Sections 4.6.2 to 4.6.4, we show how to symbolically evaluate this extracted Delta term, yielding in Section 4.6.5 a full program that computes the gradient in one go without relying on any Delta interpreter. Finally, we show in Section 4.6.6 that the Delta extraction observed in the example works in general, by an induction argument over the syntax of the core language.

4.6.1 Example

At the top of Fig. 4.19, we give a simple term t_{sc} that multiplies an array a :: Array $[n]$ $\mathbb{R}$ elementwise with its reverse and sums the result, thus computing one element of a's self-convolution. Note that a zero-dimensional array contains exactly one value, hence t_{sc} is suitable for reverse differentiation.

First, we vectorise the term by passing it through the BOT. The sumOuter in t_{sc} remains as-is, because it is not enclosed in build1 or index. The build1 part of t_{sc} gets vectorised: indexing turns into gather and the scalar primitive operation ($\times_{\mathbb{R}}$) now operates on arrays, notated ($\times_{\text{Array } sh\, \mathbb{R}}$). Of course, 'gather $[n]$ a ($\lambda[i].\ [i]$)' can be simplified to 'a', and an implementation should perform this optimisation, but we keep the term as-is throughout this section to illustrate the general case.

For (forward) differentiation, we take t_{sc} through the code transformation in Figs. 4.15 and 4.16. The result is monadic code in the IdGen monad, and is a single term $D[t'_{\text{sc}}]$ of the following type:

$$a : (\text{Array } [n]\ \mathbb{R}, \text{Delta } [n]) \vdash D[t'_{\text{sc}}] : \text{IdGen } (\text{Array } []\ \mathbb{R}, \text{Delta } [])$$

Internally, the term uses various array operations from the core language (gather, sumOuter, $\times_{\text{Array } sh\, \mathbb{R}}$) as well as pairs, let-binding, and monadic operations.

Simplification. This term could of course benefit from some simplification, most particularly using the monad laws. Formulated in **do**-notation, the properties about monads that we use are the following:

$$\begin{array}{lll}
(\textbf{do}\ x \leftarrow \textbf{return}\ E_1; E_2) & = (\textbf{do let}\ x = E_1; E_2) & \textit{(left id.)} \\
(\textbf{do}\ y \leftarrow (\textbf{do}\ x \leftarrow E_1; E_2); E_3) & = (\textbf{do}\ x \leftarrow E_1; y \leftarrow (\textbf{do}\ E_2); E_3) & \textit{(assoc.)} \\
(\textbf{do}\ y \leftarrow (\textbf{do let}\ x = E_1; E_2); E_3) & = (\textbf{do let}\ x = E_1; y \leftarrow (\textbf{do}\ E_2); E_3) & \textit{(notation)}
\end{array}$$

with E_i standing for arbitrary code. Additionally, we reorganise some let-bindings, and contract let-bindings with the same right-hand side (specifically: "a"):

$$(\textbf{let}\ x = E\ \textbf{in let}\ y = E\ \textbf{in}\ \ldots x \ldots y \ldots) = (\textbf{let}\ x = E\ \textbf{in}\ \ldots x \ldots x \ldots)$$

The result is the simplified term at the bottom of Fig. 4.19.

Note that because the original term (t_{sc}) did not have any let-bindings, its trace (the Delta term) does not have any shared subterms, and hence the Share nodes in $D[t'_{\text{sc}}]$ actually turn out to be unnecessary. In general, however, they could be needed if the source program used the result of certain subterms multiple times.

Source term:

$$
\begin{aligned}
&a : \text{Array}\ [n]\ \mathbb{R} \\
&\vdash t_{\text{sc}} = \text{sumOuter}\ (\text{build1}\ n\ (\lambda i.\ \text{index}\ a\ i \times_{\mathbb{R}} \text{index}\ a\ (n-1-i))) \\
&\quad : \text{Array}\ []\ \mathbb{R}
\end{aligned}
$$

After BOT:

$$
\begin{aligned}
t'_{\text{sc}} = \text{sumOuter}\ \big(&\text{gather}\ [n]\ a\ (\lambda[i].\ [i]) \\
&\times_{\text{Array}\ sh\ \mathbb{R}}\ \text{gather}\ [n]\ a\ (\lambda[i].\ [n-1-i])\big)
\end{aligned}
$$

After forward differentiation:

$$
\begin{aligned}
D[t'_{\text{sc}}] = \textbf{do}\ &(x, d) \leftarrow \textbf{do} \\
&(x_1, d_1) \leftarrow \textbf{do}\ (y_1, dy_1) \leftarrow \textbf{return}\ a \\
&\qquad id_1 \leftarrow \text{genID} \\
&\qquad \textbf{return}\ (\text{gather}\ [n]\ y_1\ (\lambda[i].\ [i]) \\
&\qquad\quad , \text{Share}\ id_1\ (\text{Gather}\ dy_1\ (\lambda[i].\ [i]))) \\
&(x_2, d_2) \leftarrow \textbf{do}\ (y_2, dy_2) \leftarrow \textbf{return}\ a \\
&\qquad id_2 \leftarrow \text{genID} \\
&\qquad \textbf{return}\ (\text{gather}\ [n]\ y_2\ (\lambda[i].\ [n-1-i]) \\
&\qquad\quad , \text{Share}\ id_2\ (\text{Gather}\ dy_2\ (\lambda[i].\ [n-1-i]))) \\
&id_3 \leftarrow \text{genID} \\
&\textbf{return}\ (x_1 \times_{\text{Array}\ sh\ \mathbb{R}} x_2, \text{Share}\ id_3\ (\text{Add}\ (\text{Scale}\ x_2\ d_1)\ (\text{Scale}\ x_1\ d_2))) \\
id_4 \leftarrow\ &\text{genID} \\
\textbf{return}\ &(\text{sumOuter}\ x, \text{Share}\ id_4\ (\text{SumOuter}\ d))
\end{aligned}
$$

After simplification:

$$
\begin{aligned}
D[t'_{\text{sc}}] = \textbf{do}\ &\textbf{let}\ (a_p, a_d) = a \\
&\textbf{let}\ x_1 = \text{gather}\ [n]\ a_p\ (\lambda[i].\ [i]) \\
&\textbf{let}\ x_2 = \text{gather}\ [n]\ a_p\ (\lambda[i].\ [n-1-i]) \\
&id_1 \leftarrow \text{genID}; id_2 \leftarrow \text{genID}; id_3 \leftarrow \text{genID}; id_4 \leftarrow \text{genID} \\
&\textbf{return}\ (\text{sumOuter}\ (x_1 \times_{\text{Array}\ sh\ \mathbb{R}} x_2) \\
&\qquad , \text{Share}\ id_4\ (\text{SumOuter}\ (\text{Share}\ id_3 \\
&\qquad\quad (\text{Add}\ (\text{Scale}\ x_2\ (\text{Share}\ id_1\ (\text{Gather}\ a_d\ (\lambda[i].\ [i])))) \\
&\qquad\qquad (\text{Scale}\ x_1\ (\text{Share}\ id_2\ (\text{Gather}\ a_d\ (\lambda[i].\ [n-1-i])))))))
\end{aligned}
$$

Figure 4.19: Example of the full pipeline so far.

Evaluation. To compute a gradient with this differentiated term at a certain input x : Array $[n]$ $\mathbb{R}$, run $D[t'_{\text{sc}}]$ in the environment $\{a \coloneqq (x, \mathsf{Input}\ var)\}$ for some var : DVarName $[n]$, and pass the resulting Delta term (together with the initial cotangent $[1.0]$: Array $[]$ $\mathbb{R}$) to *reversePass* from Section 4.5.

The *reversePass* function calls *backprop*, which evaluates the Delta term in the second component of the result of $D[t'_{\text{sc}}]$ in reverse, starting with its topmost Share id_4 node. The initial cotangent $[1.0]$ arrives in *eval* (Fig. 4.18) via the 'accum' map of the evaluation state (at key id_4). Evaluation then reverse-evaluates SumOuter, replicating up the initial cotangent to an n-element array which, again via the 'accum' map, gets contributed to the Add node, which propagates it on to both Scale nodes. There the array gets elementwise-multiplied with the primal arrays x_2 and x_1 before *eval* of Gather uses a 'scatter' operation to invert the index mappings and construct the two contributions to the gradient with respect to a in 'grad'. These are added together using $\vec{+}$ in the DMap.insertWith call in *eval* of Input.

Finally, the final gradient of type Array $[n]$ $\mathbb{R}$ is at the var key of the DMap returned from *reversePass*. Because the array operations in t'_{sc} operate on scalar arrays in bulk, the Delta trace contains only a few nodes, and evaluation overhead is limited.

Separated Delta term. A curious thing has happened when simplifying $D[t'_{\text{sc}}]$ to the form at the bottom of Fig. 4.19: the Delta term is fully extracted from the primal computation. The simplified $D[t'_{\text{sc}}]$ has the following structure:

$$\begin{array}{l}\textbf{do let}\ \textit{most of the primal computation}\\ \quad (id_1, \ldots, id_n) \leftarrow \textit{generate IDs}\\ \quad \textbf{return}\ (\textit{result of the primal computation}\\ \qquad\qquad , \textit{symbolic Delta term})\end{array} \tag{4.5}$$

In fact, it turns out that $D[t]$ has this structure (after simplification) for *any* term t in our core language! This is good news, because with a separated-out Delta term, we can symbolically evaluate that Delta, put the resulting symbolic gradient back in replacing the dual component of the result, and obtain a single term that computes a gradient without the need of any interpreter. In Section 4.6.6 we (informally) prove that we can always simplify $D[t]$ to the form in Eq. (4.5), but for now let us simply assume that the example in Fig. 4.19 is representative.

Let us look more closely at the structure of the dual component of the result: "*symbolic Delta term*" in Eq. (4.5). It is of course not actually a value of type Delta *sh* for some *sh*, but instead a term (call it t_d) in an extension of our core language that *produces* a Delta term. However, as we can see in Fig. 4.19, t_d in fact contains almost only Delta constructors; the exceptions are as follows:

1. The ID field in a Share constructor is a variable reference to one of the genID results.

2. The first field of a Scale constructor (the scaling constant) is a variable reference into the first **let**-part of the form. This is valid for the arithmetic operation rules in Fig. 4.16, but if there were a $D[-]$ rule for some primitive operation that puts a non-trivial term in the first field of Scale, this can be easily reduced to a variable reference by let-binding that term first.

3. Index values (in Index) and index functions (in Gather and Scatter) are terms instead of concrete values.

4. The Delta terms for the inputs, in Fig. 4.19 just a_d, may appear instead of a Delta constructor term.

Exceptions 1–3 concern positions in the Delta data type where non-Delta values are embedded, so it is to be expected that they "escape" from the strict form of a term with only Delta constructors. Exception 4 is for inputs, and if we think back to the wrapper (Section 4.1.7), we already know that these will become Input constructors. Hence we can fill in the exception-4 variable references with the appropriate Input terms, and leave just exceptions 1–3.

4.6.2 Symbolic evaluation of Delta: without proper sharing

A term that consists just of Delta constructors apart from exceptions 1–3 listed in the previous subsection is similar enough to an actual Delta term that we can symbolically evaluate it: all the information is already present for *eval* to "know what to do". To capture this particular not-quite-Delta, define a data type SymDelta (short for "symbolic delta") just like Delta from Fig. 4.13, but with the exceptional constructors modified as follows:

data SymDelta sh **where**
  Scale :: ASTVarName (Array $sh\,\mathbb{R}$) $\to$ SymDelta sh $\to$ SymDelta sh
  Share :: ASTVarName ID $\to$ SymDelta sh $\to$ SymDelta sh
  Index :: Delta $[k_1, \ldots, k_n]$ $\to$ ASTIx m $\to$ Delta $[k_{m+1}, \ldots, k_n]$
  Gather :: Delta $[k_1, \ldots, k_{m_2}, k_{m_2+1}, \ldots, k_n]$ $\to$ ASTIxFun m_1 m_2
    $\to$ Delta $[k'_1, \ldots, k'_{m_1}, k_{m_2+1}, \ldots, k_n]$
  Scatter :: Delta $[k_1, \ldots, k_{m_1}, k_{m_1+1}, \ldots, k_n]$ $\to$ ASTIxFun m_1 m_2
    $\to$ Delta $[k'_1, \ldots, k'_{m_2}, k_{m_1+1}, \ldots, k_n]$
  — Other Delta constructors with "Delta" replaced by "SymDelta"

Here we assume that the output of the $D[-]$ code transformation from Figs. 4.15 and 4.16 is represented in a data type called 'AST', and that variable references in

'AST' are represented with 'ASTVarName's. The 'ASTIx' data type is a symbolic 'Ix':

```
data ASTIx k where
  IZ   :: ASTIx 0
  (:::) :: AST (Array [] Int) → ASTIx k → ASTIx (k + 1)
```

and 'ASTIxFun k_1 k_2' similarly captures a symbolic function Ix k_1 → Ix k_2. By the observation at the end of Section 4.6.1, we can express the dual part of the return value of $D[t]$ (the "*symbolic Delta term*" part of Eq. (4.5)) as a value of type SymDelta.

The *eval* function in the non-symbolic reverse pass of Section 4.5 (Fig. 4.18) has the following type:

$$eval :: \text{Array } sh\, \mathbb{R} \to \text{Delta } sh \to \text{ES} \to \text{ES}$$

taking an incoming cotangent and a Delta term to evaluate, and working on an evaluation state of type ES. For evaluation of a SymDelta, its type would become symbolic:

$$eval :: \text{AST (Array } sh\, \mathbb{R}) \to \text{SymDelta } sh \to \text{ES} \to \text{ES}$$

in addition, of course, to changing ES as well as the type of *reversePass*. However, making *eval* evaluate symbolic Delta terms is not quite as easy as just changing its use of the core language array operations to constructors of 'AST'. Specifically, *eval* sometimes duplicates the incoming cotangent c. In Fig. 4.18, we had:

$$eval\ c\ (\text{Add}\ d_1\ d_2) = eval\ c\ d_2 \circ eval\ c\ d_1$$

Because this c is now not simply an array but instead a term that *computes* an array, passing the same c to *eval* twice will result in inserting that term twice in the gradient program.

This is work duplication, and this duplication would occur for all Add and LitArray nodes; because Add is emitted for every primitive operation with more than 1 argument, this work duplication is indeed quite egregious. For example, suppose we had the following SymDelta term, with IDs of Share nodes indicated using subscripts:[28]

$$\text{Share}_3 \to \text{Add} \circlearrowright \text{Share}_2 \to \text{Add} \circlearrowright \text{Share}_1 \to \text{Add} \circlearrowright \text{Input}$$

[28]This Delta term would arise from $D[\textbf{let } x = inp + inp \textbf{ in let } y = x + x \textbf{ in let } z = y + y \textbf{ in } z]$ with input *inp*.

Symbolically evaluating this term with an initial cotangent term c would pass "$c \vec{+} c$" to node 2,[29] "$(c \vec{+} c) \vec{+} (c \vec{+} c)$" to node 1, and "$((c \vec{+} c) \vec{+} (c \vec{+} c)) \vec{+} ((c \vec{+} c) \vec{+} (c \vec{+} c))$" to the Input node!

This is precisely the same problem as we had in Section 4.1.5: we are building a term symbolically (there a Delta term, here a program that computes a gradient), but we cannot properly express the sharing that we need. And there is no good place in *eval* to create a let-binding: *eval* puts c in the evaluation state, and passing the same c to multiple calls to *eval* just means that it ends up in multiple different places in the evaluation state.

4.6.3 Symbolic evaluation of Delta with sharing

Faced with the same problem, we apply the same solution: global sharing. Like we have Share in Delta, we add to (our extension of) the core language a new operation called 'share':

$$\text{share} : \text{ID}_{\text{AST}}\ (\text{Array}\ sh\ \rho) \to \text{Array}\ sh\ \rho \to \text{Array}\ sh\ \rho$$

where ID_{AST} is a type like ID, except that (1) it is indexed by the full type of the array instead of just the shape, and (2) it refers to a primal program fragment of 'AST' type, not a Delta term fragment.

In its semantics on normal, concrete arrays, share does nothing: it simply returns its second argument. However, in an AST describing a program, it indicates sharing: two terms wrapped by share nodes with the same ID_{AST} are equal and must be computed only once.

To solve the problem from the previous subsection, we must ensure that any cotangent terms that *eval* duplicates are "protected" by share. This can be done in two different ways:

1. Pessimistically, i.e. the same way we placed Share constructors in $D[-]$: whenever we construct a non-trivial cotangent, protect it against possible later duplication. This means that in *eval* in Fig. 4.18, all recursive calls that do not simply pass on "c" as the first argument would be changed to first generate an ID_{AST} with $\text{genID}_{\text{AST}}$, and then instead of calling $eval\ (\ldots c \ldots)\ d$, call $eval\ (\text{share}\ id\ (\ldots c \ldots))\ d$), where id is the generated ID_{AST}. To support the case where the initial cotangent passed to *reversePass* is non-trivial, *reversePass* would also wrap the initial cotangent in share before passing it to *eval*.

2. By optimistically assuming that cotangents may not be duplicated at all, and to add share wrappers when duplicating cotangents instead. This means

[29] The broadcasted additions here ("$\vec{+}$") come from the use of ($\vec{+}$) in *eval* of Share in Fig. 4.18.

generating an $\mathrm{ID}_{\mathrm{AST}}$ in the right-hand side of *eval* for Add and LitArray, and to pass share-wrapped cotangents (with the same $\mathrm{ID}_{\mathrm{AST}}$!) to each recursive call to *eval* there.

With both approaches, the reverse pass will need to run in the IdGen monad: as we will see in Section 4.6.4, just like with Share in Section 4.1.5, we need the generated IDs to be strictly monotonically increasing, so that subterms always have lower IDs. This invariant is upheld by both (1) and (2).

While both approaches are valid and allow us to preserve all sharing, each has advantages and disadvantages. Placing share pessimistically as in (1), we may end up placing many unnecessary share nodes if the program never actually duplicates values. When placing them optimistically as in (2), a large tree of Add nodes produces many internal share nodes on the cotangents, even if no actual computation happens on the cotangents in between and hence there is nothing of worth to deduplicate.

When placing Share nodes to encode sharing of Delta terms in Section 4.1.5, we were obliged to choose approach (1): since the user program can pass around and compute with dual numbers (and hence, indirectly, pass around Delta terms) without us knowing precisely where duplication happens, we could not share at duplication sites only. With dual arrays in Section 4.5, because we have a restricted input language, we *can* see all duplication (namely, when a let-bound variable is used multiple times), but for consistency we kept using approach (1). Here, we instead choose the optimistic approach (2), not because we cannot use (1), but because it results in a much smaller change to *eval*: only in the equations for Add and LitArray.

The resulting reverse pass is shown in Fig. 4.20. The ES data type and *reversePass* are adapted for AST-typed cotangents from Section 4.5.4; *backprop* is still unchanged from Fig. 4.5, except for substituting "DMap" for "Map". The *eval* function is modified from Fig. 4.18 to share the incoming cotangent whenever it would be duplicated in multiple sub-evaluations.

It should be noted that in Fig. 4.20 (the symbolic reverse pass), we use some convenient notational punning to emphasise the similarity to Fig. 4.18 (the non-symbolic array reverse pass): in Fig. 4.18, the core language operations (gather, index, etc.) referred to *array operations* that were performed on concrete arrays. In Fig. 4.20, they instead refer to AST constructors of the gradient term that is being built up.

Notable is that we do *not* need to wrap share around cotangents that are added to the cotangent accumulation map (the 'accum' field of ES) or the gradient accumulation map (the 'grad' field): they are never duplicated, because they are in fact used exactly once (either to retrieve their final value or to be added to yet another contribution).

```
data ES = ES
  { grad :: DMap DVarName (λsh. AST (Array sh ℝ))
  , dfrag :: DMap ID SymDelta
  , accum :: DMap ID (λsh. AST (Array sh ℝ)) }

reversePass :: AST (Array sh ℝ) → SymDelta sh
            → DMap DVarName (λsh. AST (Array sh ℝ))
reversePass c d = grad (backprop (eval c d (ES {} {} {})))

backprop :: ES → ES
backprop s = case DMap.maxViewWithKey (accum s) of
               Just ((i, c), acc') →
                 let d = dfrag s DMap.! i
                     s' = eval c d (s { accum = acc'
                                      , dfrag = DMap.delete i (dfrag s) })
                 in backprop s'
               Nothing → s

eval :: AST (Array sh ℝ) → SymDelta sh → ES → IdGen ES
eval c Zero          = return
eval c (Input v)     = λs. return (s { grad = DMap.insertWith (+) v c (grad s) })
eval c (Add d1 d2)   = λs. do id ← genID_AST
                              let cShared = share id c
                              eval cShared d1 s >>= eval cShared d2
eval c (Scale arr d) = eval (c ⋆ arr) d
eval c (Share i d)   = λs. return (s { dfrag = DMap.insert i d (dfrag s)
                                     , accum = DMap.insertWith (+) i c (accum s) })
eval c (Index d i)   = eval (oneHot (shapeDelta d) i c) d
eval c (SumOuter d)  = eval (replicate (head (shapeDelta d)) c) d
eval c (Gather d f)  = eval (scatter (shapeDelta d) c f) d
eval c (Scatter d f) = eval (gather (shapeDelta d) c f) d
eval c (LitArray ds) = λs. do id ← genID_AST
                              let cShared = share id c
                              let [n] = shape ds
                              eval (index cShared [n - 1]) (index ds [n - 1]) s
                                >>= ··· >>= eval (index cShared [0]) (index ds [0])
eval c (Replicate d)   = eval (sumOuter c) d
eval c (Transpose_{j1,...,jm} d) = eval (tr_{inversePermutation(j1,...,jm)} c) d
eval c (Reshape d)     = eval (reshape (shapeDelta d) c) d
```

Figure 4.20: The symbolically-executing reverse pass. Notational punning: the core language operations (gather, index, etc.) denote AST constructors here.

Example. Let s be the symbolic Delta term extracted from the simplified version of $D[t'_{\text{sc}}]$ in Fig. 4.19, with a_d replaced with `Input "inp"`. If we run *reversePass* from Fig. 4.20 on it, we get the following:

$$\begin{aligned}&\mathit{reversePass}\ [1.0]\ s = \\ &\quad \{\ \texttt{"inp"} \mapsto \mathsf{scatter}\ [n]\ (x_2 \mathbin{\vec{\cdot}} \mathsf{share}\ 1\ (\mathsf{replicate}\ n\ [1.0]))\ (\lambda[i].\ [i])\ \vec{+} \\ &\qquad\qquad \mathsf{scatter}\ [n]\ (x_1 \mathbin{\vec{\cdot}} \mathsf{share}\ 1\ (\mathsf{replicate}\ n\ [1.0]))\ (\lambda[i].\ [n-1-i])\ \}\end{aligned}$$

The first argument to *reversePass* is the incoming cotangent, which in this case is a zero-dimensional array of scalars (i.e. a single scalar).

Note that the '1' in the share nodes was generated by the $\text{genID}_{\text{AST}}$ call in *eval* of the Add node. Further, because an interpreter of this term is supposed to memoise the results of evaluating share-wrapped subterms, a proper interpreter would *not* execute the replicate (the reverse derivative of the sumOuter in t_{sc}) twice. To avoid excessive memory use in representing this AST in a compiler, one should ensure that share-wrapped terms are actually shared in-memory in the compiler too.

4.6.4 Converting global sharing to let-bindings

In principle, we can just generate the final gradient program by looking back at the structure of $D[t]$ after simplification (Eq. (4.5) on page 171), and replacing "*symbolic Delta term*" with 'reconstruct' (Fig. 4.6) applied to the output of *reversePass* (Fig. 4.20) on the original "*symbolic Delta term*". However, if we do this, the result is a term that has both let-bindings *and* global sharing using share: the let-bindings occur in the primal computation, and the global sharing is in the dual computation (produced by the symbolic reverse pass). Because global sharing is rather seldomly used in compilers, for compiling this gradient program to native code it is likely necessary to ensure that the whole term uses let-bindings for sharing, and nothing else.

Thus, we need to eliminate share from the term produced by the (wrapper around the) symbolic reverse pass. Fortunately, this term only contains a very limited set of constructs:

- It contains variable references: these come from the first field of Scale, which we restricted to be a variable reference in Section 4.6.1.
- It contains the initial cotangent passed to *reversePass*, which we will assume to be either a constant literal or a variable reference. (The wrapper in Section 4.6.5 will put a variable reference here.)
- Otherwise, it contains only what *eval* from Fig. 4.20 produces explicitly. For our core language, one can verify (by closely reading Fig. 4.20) that

this is limited to: share, $(\vec{+})$, $(\vec{\cdot})$, oneHot, replicate, scatter, gather, index, sumOuter, tr, and reshape. Some subtleties:

- The index construct contains a list of terms for the position to index at; these are arbitrary terms from the primal program.
- Similarly, the gather and scatter constructs contain index functions, which also come straight from the primal program.

These components may contain arbitrary terms, but because *eval* just preserves these terms as-is, we are sure that they do not contain share, meaning that we can also keep them as-is here.

In particular, *it does not contain let-bindings outside of untouched, copied subterms.* (There may of course be internal let-bindings inside of the index and index-function arguments to index and gather/scatter, but we do not need to care about those.) Because the semantics of share is somewhat murky in the presence of let-bindings, their absence makes our task of conversion to standard let-bindings much more straightforward.

The conversion function, for this peculiar language (the output of symbolic *eval*) that we need to support, is given in Fig. 4.21. In this figure, we indicate meta-'let' (i.e. a let-binding in the language that *unshare* etc. themselves are written in) by "**let**", and a term-'let' in the AST data type by "**let**". All 'let's in *unshare* are meta; the one 'let' in *stackLets* is a term.

The conversion function from global sharing to let-bindings is *shareToLet*, which first collects all shared term fragments in a DMap using *unshare*, and then stacks **let** terms on top of the root term fragment using *stackLets*. The *unshare* function takes a term t with share nodes inside and returns a dictionary of all the fragments inside t, together with a single root term fragment t' that consists of the constructors near the root of t above the first share nodes. In both halves of the return value of *unshare*, share-wrapped subterms have been replaced with fresh variable names.

Thus, inside *unshare*, whenever we encounter a share node, we have to decide on a variable name to represent this fragment in the unshared term. However, all we have is an $\mathrm{ID}_{\mathrm{AST}}$. If we have external knowledge that $\mathrm{ID}_{\mathrm{AST}}$ and ASTVarName refer to equal (or convertible) types, and that the ASTVarName derived from an $\mathrm{ID}_{\mathrm{AST}}$ in this fashion will never be used by the user or generated by the machinery before this section, the side-effect "⟨generate a fresh variable name⟩" in *unshare* could simply convert i to a variable name instead. If not, then *unshare* should additionally run in a simple state monad for generating fresh names.

Having a DMap of all term fragments of t, we simply build a long stack of let-bindings on top of the root term fragment. For this to make sense, we need to ensure that the fragments without dependencies are bound at the top of the stack,

```
unshare :: DMap ID_AST (λτ. (ASTVarName τ, AST τ)) → AST τ
        → (DMap ID_AST (λτ. (ASTVarName τ, AST τ)), AST τ)
unshare m x                = (m, x)        — variable references
unshare m c                = (m, c)        — constant literals
unshare m (share i t)      = case DMap.lookup i m of
                               Just (var, _) →    — return a variable reference term
                                 (m, var)
                               Nothing →
                                 — These are meta-'let's, not AST constructors
                                 let var = ⟨generate a fresh variable name⟩
                                     (m', t') = unshare m t
                                 in (DMap.insert i (var, t') m', var)
unshare m (s +⃗ t)         = let (m1, s') = unshare m s      — idem
                                 (m2, t') = unshare m1 t
                             in (m2, s' +⃗ t')
unshare m (s ·⃗ t)         = let (m1, s') = unshare m s      — etc.
                                 (m2, t') = unshare m1 t
                             in (m2, s' ·⃗ t')
unshare m (oneHot sh i t)  = let (m', t') = unshare m t in (m', oneHot sh i t')
unshare m (replicate k t)  = let (m', t') = unshare m t in (m', replicate k t')
          — Note: f in scatter/gather and ix in index remain untouched
unshare m (scatter sh t f) = let (m', t') = unshare m t in (m', scatter sh t' f)
unshare m (gather sh t f)  = let (m', t') = unshare m t in (m', gather sh t' f)
unshare m (index t ix)     = let (m', t') = unshare m t in (m', index t' ix)
unshare m (sumOuter t)     = let (m', t') = unshare m t in (m', sumOuter t')
unshare m (tr_{k1,...,kn} t) = let (m', t') = unshare m t in (m', tr_{k1,...,kn} t')
unshare m (reshape sh t)   = let (m', t') = unshare m t in (m', reshape sh t')

stackLets :: DMap ID_AST (λτ. (ASTVarName τ, AST τ)) → AST τ → AST τ
stackLets m t = case DMap.maxViewWithKey m of
                  Just ((_, (var, t')), m') →
                    stackLets m' (let var = t' in t)    — An AST let-binding!
                  Nothing → t

shareToLet :: AST τ → AST τ
shareToLet t = let (m, t') = unshare {} t in stackLets m t'
```

Figure 4.21: Converting global sharing to let-bindings.

the fragments that depend just on those come right after, etc. Fortunately, because the IDs are generated monotonically in the symbolic *eval* function (Fig. 4.20), it suffices to simply bind the IDs from lowest to highest. Because *stackLets* builds the stack from the bottom up instead of from the top down, it starts with the term with the *highest* ID at the bottom of the stack, and then proceeds upwards with lower and lower IDs until the whole DMap is exhausted.

Example. Running *unshare* on the gradient term (with share) that we derived for the example at the end of Section 4.6.3, we get:

$$
\begin{aligned}
&\mathit{unshare}\ \{\}\ (\mathit{reversePass}\ [1.0]\ s) = \\
&\quad \big(\{\ 1 \mapsto (\texttt{"shared1"}, \mathsf{replicate}\ n\ [1.0])\ \} \\
&\quad , \mathsf{scatter}\ [n]\ (x_2 \vec{\cdot}\ \mathit{shared1})\ (\lambda[i].\ [i])\ \vec{+} \\
&\qquad \mathsf{scatter}\ [n]\ (x_1 \vec{\cdot}\ \mathit{shared1})\ (\lambda[i].\ [n-1-i])\big)
\end{aligned}
$$

Here we generated the name "shared1" for the ID 1. Completing with *stackLets* (which has a rather easy task in this case with just one shared fragment), we get a term without share:

$$
\begin{aligned}
&\mathit{shareToLet}\ (\mathit{reversePass}\ [1.0]\ s) = \\
&\quad \textbf{let}\ \mathit{shared1} = \mathsf{replicate}\ n\ [1.0] \\
&\quad \textbf{in}\ \mathsf{scatter}\ [n]\ (x_2 \vec{\cdot}\ \mathit{shared1})\ (\lambda[i].\ [i])\ \vec{+} \\
&\qquad \mathsf{scatter}\ [n]\ (x_1 \vec{\cdot}\ \mathit{shared1})\ (\lambda[i].\ [n-1-i])
\end{aligned}
$$

4.6.5 Wrapper

Now we have all the components to assemble the final algorithm in a wrapper that the user can make sense of. A pseudocode rendering of this wrapper is shown in Fig. 4.22.

All stages of the pipeline come to the fore here:

- We start with a term t :: AST (Array [] $\mathbb{R}$) that returns a single scalar; t has a number of free variables that constitute the input parameters that we differentiate with respect to.
- We first vectorise t to use bulk operations using the BOT (Section 4.4); this results in t_{bulk}.
- We (forward-)differentiate t_{bulk} using $D[-]$ (Section 4.5), and rebind its free variables: the **let** in the assignment to t_{diff} is an embedded let-binding. Where t_{bulk} (still) had free variables x_i : Array sh_i ρ_i, the differentiated term $D[t_{\text{bulk}}]$ has free variables x_i : (Array sh_i ρ_i, Delta sh_i). The **let**, which

$$
\begin{array}{l}
\textit{— First argument has free variables: } x_1 : \text{Array } sh_1\ \rho_1, \ldots, x_n : \text{Array } sh_n\ \rho_n \\
\textit{— Result has free vars.: } x_1 : \text{Array } sh_1\ \rho_1, \ldots, x_n : \text{Array } sh_n\ \rho_n, c : \text{Array } [] \ \mathbb{R} \\
\text{wrapper} :: \text{AST (Array } [] \ \mathbb{R}) \\
\qquad \rightarrow \text{AST (Array } [] \ \mathbb{R}, (\text{Array } sh_1\ \rho_1, \ldots, \text{Array } sh_n\ \rho_n)) \\
\text{wrapper } t = \\
\quad \textbf{let } t_{\text{bulk}} = \text{BOT}[t] \\
\qquad t_{\text{diff}} = \textbf{let } (x_1, \ldots, x_n) = ((x_1, \text{Input } 1), \ldots, (x_n, \text{Input } n)) \textbf{ in } D[t_{\text{bulk}}] \\
\qquad \begin{pmatrix} \textbf{do let } \textit{primalBinds} \\ \quad id_1 \leftarrow \text{genID}; \ldots; id_m \leftarrow \text{genID} \\ \quad \textbf{return } (t_1, t_2) \end{pmatrix} = \text{extractBySimplification}[t_{\text{diff}}] \\
\qquad t_2' = \textit{shareToLet } (\textit{reversePass } c\ (t_2[id_1 := 1, \ldots, id_m := m])) \\
\quad \textbf{in } \begin{pmatrix} \textbf{let } \textit{primalBinds} \\ \textbf{in } (t_1, t_2') \end{pmatrix}
\end{array}
$$

Figure 4.22: The wrapper for the full algorithm. See Section 4.6.5 for details on the notation.

should be read as a *non-recursive* let-binding, provides the second components of those and ensures that t_{diff}'s free variables have type Array $sh_i\ \rho_i$ again.

- Then we simplify as in the example (Fig. 4.19 in Section 4.6.1), and as more rigorously justified in Section 4.6.6. This produces a term of a specific form (Eq. (4.5)); we pattern-match out the components: *primalBinds*, m, id_i, t_1 and t_2. (The notation between the large parentheses is a *term*.)

- Finally, in the assignment to t_2', we first substitute $1, \ldots, m$ for $id_1, \ldots, id_m$ in t_2 (essentially "running the IdGen monad" in poor man's fashion), and then put the result through the machinery of Sections 4.6.3 and 4.6.4. The "c" here is a *term*: a variable reference to the variable 'c', the initial cotangent. This means that t_2' has as free variables:

  - The primal inputs x_i;
  - Any names bound in *primalBinds*;
  - The initial cotangent: c : Array $[]\ \mathbb{R}$.

- The result of the wrapper is a term (again written between large parentheses) that first runs the part of the primal that was shared between t_1 and t_2, and then returns a pair of the primal result (t_1) and the gradient that is, by now, a standard term (t_2'). The result of the wrapper computes not only

a gradient, but also the primal result, as can be seen in its type; the free variables of this term are the inputs x_i as well as the initial cotangent c.

4.6.6 Delta extraction works in general

In Section 4.6.1 we saw that the derivative of the example term, $D[t'_{\text{sc}}]$, simplified to a particularly useful form:

$$\begin{array}{l}\textbf{do let } \textit{most of the primal computation} \\ \quad (id_1, \ldots, id_n) \leftarrow \textit{generate IDs} \\ \quad \textbf{return } (\textit{result of the primal computation} \\ \qquad\quad , \textit{symbolic Delta term})\end{array} \tag{4.5 again}$$

and we claimed that this in fact holds for *all* source terms. Furthermore, we specified that the "*symbolic Delta term*" had to conform to some requirements:

1. The ID field of a Share constructor is a variable reference;
2. Similarly for the scaling field of a Scale constructor;
3. The index (function) fields of Index, Gather and Scatter can be arbitrary terms;
4. Otherwise, it consists of only Delta constructors, except for a variable reference to a Delta component of an input.

We can prove that this form holds for the output of $D[-]$ on our core language by induction. To do so, we look at every equation in Figs. 4.15 and 4.16. We will consider only source terms of type Array $sh\ \mathbb{R}$ for some shape sh; of course, such a program may also contain subterms of type Array $sh\ \rho$ with ρ unequal to $\mathbb{R}$, but due to the structure of the equations in the transformation, we end up being able to ignore those subterms.

Observe that for constructs of type Array $sh\ \mathbb{R}$ in Figs. 4.15 and 4.16, the right-hand side of the equation fits the following form:[30]

$$\begin{array}{l}\textbf{do } (x_1, d_1) \leftarrow D[t_1]; \ldots; (x_n, d_n) \leftarrow D[t_n] \\ \quad \textbf{let } \textit{most of the primal computation} \\ \quad (id_1, \ldots, id_m) \leftarrow \textit{generate IDs} \\ \quad \textbf{return } (\textit{result of the primal computation} \\ \qquad\quad , \textit{symbolic Delta term})\end{array} \tag{4.6}$$

The t_i are direct subterms of the term on the left-hand side of the equation that are also of scalar-array type, and the "*symbolic Delta term*" adheres to the same

[30] For the polymorphic constructs x, cond and **let**, one can artificially distinguish the $\mathbb{R}$ and non-$\mathbb{R}$ cases and subsequently look only at the $\mathbb{R}$ case.

constraints as we set for Eq. (4.5) above, *except* that it may also refer to the d_i with variable references *at most once*. The requirement that each d_i is referred to at most once prevents us from needing to introduce global sharing here just yet.

Now assuming (by the induction hypothesis) that $D[t_i]$ is already in form Eq. (4.5), our only task is to rewrite Eq. (4.6) to Eq. (4.5). If we do so, then by induction, form Eq. (4.5) can be derived from the derivative of every source program term by repeated simplification, and we are done.

But indeed, this is not very difficult: subscripting the components of Eq. (4.5) with i according to which $D[t_i]$ it corresponds to, the nested structure looks as follows:

$$
\begin{aligned}
&\textbf{do } (x_1, d_1) \leftarrow \textbf{do let } (\textit{most of the primal computation})_1 \\
&\qquad\qquad\qquad\qquad (id_1, \ldots, id_{m_1}) \leftarrow \textit{generate IDs} \\
&\qquad\qquad\qquad\qquad \textbf{return } ((\textit{result of the primal computation})_1 \\
&\qquad\qquad\qquad\qquad\qquad\qquad , (\textit{symbolic Delta term})_1) \\
&\quad \ldots \\
&\quad (x_n, d_n) \leftarrow \textbf{do let } (\textit{most of the primal computation})_n \\
&\qquad\qquad\qquad\qquad (id_1, \ldots, id_{m_n}) \leftarrow \textit{generate IDs} \\
&\qquad\qquad\qquad\qquad \textbf{return } ((\textit{result of the primal computation})_n \\
&\qquad\qquad\qquad\qquad\qquad\qquad , (\textit{symbolic Delta term})_n) \\
&\quad \textbf{let } \textit{most of the primal computation} \\
&\quad (id_1, \ldots, id_m) \leftarrow \textit{generate IDs} \\
&\quad \textbf{return } (\textit{result of the primal computation} \\
&\qquad\qquad , \textit{symbolic Delta term})
\end{aligned}
$$

which is easily rewritten to:

$$
\begin{aligned}
&\textbf{do let } (\textit{most of the primal computation})_1 \\
&\quad \ldots \\
&\quad (\textit{most of the primal computation})_n \\
&\quad x_1 = (\textit{result of the primal computation})_1 \\
&\quad \ldots \\
&\quad x_n = (\textit{result of the primal computation})_n \\
&\quad \textit{most of the primal computation} \\
&(id_1^1, \ldots, id_{m_1}^1, \ldots, id_1^n, \ldots, id_{m_n}^n, id_1, \ldots, id_m) \leftarrow \textit{generate IDs} \\
&\textbf{return } (\textit{result of the primal computation} \\
&\qquad , (\textit{symbolic Delta term})[d_1 := (\textit{symbolic Delta term})_1, \ldots, \\
&\qquad\qquad\qquad\qquad\qquad\qquad d_n := (\textit{symbolic Delta term})_n])
\end{aligned}
$$

with some alpha-renaming for the *id* variables, and potentially other variables to avoid clashing names. Note that we are allowed to simply substitute d_i into the symbolic Delta term because each occurs at most once, as noted above.

This completes the induction, and confirms that the example in Fig. 4.19 at the beginning of this section was indeed representative.

4.7 Implementation

We have a Haskell implementation of the algorithms described in this chapter.[31] The implementation contains various features not described in detail in the chapter:

- The source language of the library is a shallowly embedded array language in Haskell; the resulting staging features are described in Section 4.10.2.

- Dual-numbers AD admits a very elegant implementation in a functional language via type classes [Elliott 2009]; dual-numbers reverse AD as presented in [Krawiec et al. 2022] and Chapter 3 preserves this property to an extent, although this property was not emphasised. The possibility of a modular implementation based on type classes remains true for the core AD algorithm in this chapter (Section 4.5); the BOT naturally cannot, but staging (Section 4.10.2) can save us here.

  Our implementation explicitly revives the type class approach and contains one implementation of AD (from which the dual arrays algorithm of Section 4.5 and the naive algorithm of Section 4.1 are special cases) and one implementation of staging (underlying both the staging of the source program and the symbolic evaluation in Section 4.6.3). These are all instances of a type class modelling our core array language. Details about this system and the sharing-related subtleties that need to be solved are given in Sections 4.10.3 and 4.10.4.

- As briefly mentioned before, the implementation supports a higher-order 'fold' operation with the restriction that the combination function must be closed; this restriction is not imposed on 'build1', which allows open element functions. Furthermore, the implementation has full support for binary tuples (i.e. pairs) in the source language and limited support for *regular* nested arrays: after struct-of-arrays transformation, all component arrays must be fully rectangular. That is to say: nested array support cannot be use to get around the prohibition of jagged arrays.

The library benchmarks favourably against `ad`[32], but performance competitive with state-of-the-art machine learning toolkits is future work.

[31] Available as a package here: `https://hackage.haskell.org/package/horde-ad`

[32] `https://hackage.haskell.org/package/ad`

4.8 Discussion and future work

Some of the simplicity and easy generalisation of dual-numbers AD is retained in our algorithm: we describe the actual AD component of Section 4.5 to work only on the output of the BOT, i.e. a quite restricted language of bulk array operations — also to make the compile-time differentiation trick of Section 4.6 work — but the AD code transformation itself ($D[-]$) would not care if we added e.g. product types, sum types or function types to the language. The role of the scalar type $\mathbb{R}$ is now fulfilled by *arrays* of scalars (Array $sh\ \mathbb{R}$), but the structure of the algorithm is the same.

The other parts of the algorithm (the BOT (Section 4.4) and symbolic evaluation (Section 4.6)) are not so kind; especially proper dynamic control flow (i.e. a lazy conditional, loops, perhaps recursion) makes it unclear how to vectorise, and throws a wrench in the rather straightforward Delta extraction process of Section 4.6.6. Paszke et al. [2021a] are able to "unzip" the primal computation from the dual computation even in the presence of dynamic control flow, and they do so by evaluating the conditional *twice*: once in the primal, where they store and "export" intermediate values from the computation in the branch taken to outside the conditional, and once in the dual (analogous to our Delta evaluation), where they make use of the stored conditional boolean as well as the stored intermediate values to run the reverse pass of the correct branch. Perhaps an extension to Delta could allow such tricks, but it is unclear how precisely.

From a performance perspective, the BOT is rather uncomfortable: *array fusion* (reducing the number of "loops" / passes over the data; see Section 2.1.4) is typically seen as an optimisation by reducing loop overhead and memory traffic, but the BOT does the exact opposite thing. As we saw in Section 4.6, the primal program is mostly preserved in the simplified differentiated version; the many additional intermediate values stored in variables seem to inhibit re-fusion of the generated primal code, but in fact, with sufficiently clever fusion algorithms, this might not need to be the case [van Balen et al. 2024]. However, the reverse pass is generally more expensive than the primal pass in reverse AD (simply because it involves more computation [Griewank and Walther 2008]), and there it is less clear to what extent fusion opportunities are preserved through differentiation. We would like to get a better understanding of the interaction between vectorisation, differentiation and fusion (and other array-program performance optimisations).

Finally, as briefly alluded to earlier, our implementation has experimental support for a sequential version of the 'fold' SOAC, albeit with the restriction that the combination function must be closed. We suspect that this restriction can be removed, while retaining the possibility to perform the Delta extraction of Section 4.6. Furthermore, this approach may extend to other SOACs — including 'build'! — but it sacrifices the dual-numbers heritage of the algorithm. The details

are yet unclear, as is the value of such a fundamental change to the algorithm.

In conclusion, the AD algorithm presented in this chapter has various desirable properties, but more research is needed to extract all of its value and make it competitive with the state-of-the-art.

4.9 Related work

Automatic differentiation. The actual AD algorithm in this chapter (excluding preprocessing in the BOT) takes the form "forward-differentiate (Section 4.5 ($D[-]$)), unzip primal from dual (Section 4.6.6), transpose forward derivative to reverse derivative (Section 4.6.3 (*eval*))"; this separation was already implicitly present in [Krawiec et al. 2022] (which we build on) and Chapter 3, but it becomes much more explicit in this chapter with symbolic transposition. This three-part structure is also used in Dex [Paszke et al. 2021a] and further explained, albeit on a language without arrays or dynamic control flow, by Radul et al. [2023]. In Dex, the `for` syntax (their equivalent of build) is differentiated by introducing mutable accumulators in the derivative program; this allows array indexing (our index) to have an efficient derivative while still computing dense cotangents. The purely functional approach in this chapter avoids such pervasive mutability.

CHAD, discussed in detail in Chapters 5 to 7, does not exhibit this three-part structure.

Vectorisation. Aggressive vectorisation (unfusion) as performed by our BOT is uncommon in prior work. However, a similar kind of "vectorising map" can be found in JAX [Bradbury et al. 2018] as `jax.vmap` and as a prototype feature in PyTorch [Paszke et al. 2017] as `torch.vmap`. The PyTorch implementation shares our restriction that array shapes must be statically known. These implementations are primarily for expressivity or ease of writing models. Futhark [Henriksen et al. 2017, §5.1] employs a vectorisation-like transform for making code more suitable for GPU compilation, by making more parallelism statically visible; we go further in that we do not only vectorise what is easy to vectorise, but we aggressively vectorise the entire expression.

Vectorisation in the presence of unequal array shapes traditionally requires some form of flattening; see, for example, NESL [Blelloch 1992] or Data-Parallel Haskell [Chakravarty et al. 2007]. This flattening tends to give non-negligible runtime overheads; we elected to avoid these overheads by simply not supporting unequal array shapes in vectorised code.

Acknowledgements. We would like to thank Tom Ellis for good discussion and reflection on algorithms and presentation.

4.10 Appendices

4.10.1 Generalisation of the output type

The top-level interface to our AD algorithm as finalised in Section 4.6, shown in Fig. 4.22, has the following type when seen as a code transformation:

$$
\begin{aligned}
&x_1 : \text{Array } sh_1\ \rho_1, \ldots, x_n : \text{Array } sh_n\ \rho_n \\
&\quad \vdash t : \text{Array } [] \ \mathbb{R} \\
&\qquad \rightsquigarrow \\
&x_1 : \text{Array } sh_1\ \rho_1, \ldots, x_n : \text{Array } sh_n\ \rho_n, c : \text{Array } [] \ \mathbb{R} \\
&\quad \vdash \text{wrapper}[t] : (\text{Array } [] \ \mathbb{R}, (\text{Array } sh_1\ \rho_1, \ldots, \text{Array } sh_n\ \rho_n))
\end{aligned}
$$

While this is typically sufficiently general in the input type, some applications may require more complex output types than a single real scalar.

Computing the Jacobian matrix for such a more general function requires multiple passes with reverse AD. (Forward AD cares little about the size of the output, and instead requires multiple passes if the *input* consists of multiple scalars.) If nested arrays are supported, one can generalise to an array of output scalars straightforwardly:

$$
\begin{aligned}
&x_1 : \text{Array } sh_1\ \rho_1, \ldots, x_n : \text{Array } sh_n\ \rho_n, c : \text{Array } sh\ \mathbb{R} \\
&\quad \vdash \textbf{let } r = \mathsf{build}\ (\mathsf{shape}\ c)\ (\lambda ix.\ \textbf{let } c = \mathsf{index}\ c\ ix\ \textbf{in}\ \text{wrapper}[\mathsf{index}\ t\ ix]) \\
&\quad\ \ \textbf{in } (\mathsf{map}\ (\lambda x.\ \text{toScalar}\ (\text{fst}\ x))\ r, \mathsf{map}\ (\lambda x.\ \text{snd}\ x)\ r) \\
&\quad : (\text{Array } sh\ \mathbb{R}, \text{Array } sh\ (\text{Array } sh_1\ \rho_1, \ldots, \text{Array } sh_n\ \rho_n))
\end{aligned}
$$

where:

$$
\begin{aligned}
&\mathsf{map}\ f\ x = \mathsf{build}\ (\mathsf{shape}\ x)\ (\lambda ix.\ f\ (\mathsf{index}\ x\ ix)) \\
&\text{toScalar} :: \text{Array } []\ \tau \to \tau
\end{aligned}
$$

In other words: run the normal algorithm for every scalar in the output, and collect the results. This approach also generalises conceptually to more complicated output types involving e.g. tuples, but the resulting transformation becomes rather notation-heavy and is omitted here.

4.10.2 Staging (embedding in Haskell)

One of the goals of our library is to be a *library*: not only is it a hassle for a user to introduce additional code preprocessors or compiler plugins into their workflow in order to use a nice AD algorithm for array programs, it would also be a higher maintenance burden for us: a code preprocessor must diligently stay up to date with the latest changes and additions to the language syntax, and a compiler plugin must stay up to date with ever-changing compiler internals. A library exposing an embedded language does not have these problems. An

additional advantage of implementing an embedded language, as compared to a separate language with a distinct compilation toolchain, is that it is easier to expose smaller steps of the compilation process to the user, allowing them to essentially customise and assemble their compiler. Despite this flexibility, type-safety of the compiler as well as with the user's other code is maintained by simply using the typechecker of the host language — in our case, Haskell.

Because we want to do a non-local code transformation (the BOT, which we explain in detail in Section 4.4) on the program written by the user, we need a syntax tree of the embedded program — in other words, we need a *deep embedding*. This automatically means that we get a level of staging in the interface to the library: when the user-written Haskell code runs, it generates code that gets interpreted by our library (`horde-ad`). In particular, instead of doing computation, every library method (that is part of the array interface) constructs a small bit of an *abstract syntax tree (AST)*. The staging that we get this way also allows the user to perform various kinds of meta-programming without us having to do anything for it; the downside is that when the user is writing their program, they have to be aware of this staging step, and that they have to make an explicit decision about what code is meta-programming and what code is embedded. Staging in `horde-ad`, implemented with type classes and understood in terms of universal algebra, is described in depth in Section 4.10.3.

Static control flow. An important example of this meta-programming is *static control flow*: control flow that does not end up in the program to be differentiated, but can only depend on statically-known parameters. Such not-quite-dynamic control flow is common in probabilistic programming and machine learning. By partially evaluating it away before interpreting the code as a program to be differentiated, we can express e.g. loop unrolling, or assembling model components from various bits and pieces depending on external information. As an example of loop unrolling, consider the following Haskell code:[33]

$$
\begin{aligned}
&\mathit{sillyAlt} \;::\; \text{Int} \to \text{Array 1 Float} \to \text{Array 1 Float}\\
&\mathit{sillyAlt}\ 0\ v = v\\
&\mathit{sillyAlt}\ n\ v \mid \text{even } n \quad\;\; = \text{rmap}\ (\backslash x \to 2 \cdot x)\ (\mathit{sillyAlt}\ (n-1)\ v)\\
&\qquad\qquad\;\; \mid \text{otherwise} = \text{rmap}\ (\backslash x \to x + 1)\ (\mathit{sillyAlt}\ (n-1)\ v)
\end{aligned}
$$

Recall that the library-provided functions build up a small bit of AST instead of doing actual computation. Thus, the function *sillyAlt* builds up a computation consisting of n layers, each here an elementwise 'rmap', `horde-ad`'s function to

[33] Here we stick to the simplified Array $n\ \tau$ notation from the rest of the chapter, but in a real Haskell code using the `horde-ad` library we would write the same as Concrete (TKR $n\ \tau$)

map a function elementwise over an array.[34] In contrast to Array (by which we mean the type of embedded arrays in `horde-ad`), Int is not an embedded type, thus all that `horde-ad` sees is various invocations of rmap nested inside each other.

Because Haskell-native operations (i.e. not from the library, such as plain `if`-expressions) only type-check on meta-values, not embedded values, and vice-versa embedded operations only type-check on embedded values, whether an operation is staged is fully apparent from the types of the values being operated on. An 'Int' is not staged, but an 'Array 0 Int' — i.e. a zero-dimensional array of integers and hence also representing a single integer — is staged because 'Array' is an embedded type.

Dynamic control flow. In addition to static control flow, which is evaluated away in staging, `horde-ad` supports a limited form of dynamic control flow: strict conditionals, or selections. (Loops of statically unknown length are currently unsupported due to the difficulty in handling them in the BOT (Section 4.4) and in AD (Section 4.5).) These dynamic conditionals are exposed via an *embedded* if-expression, which takes an embedded boolean expression and embedded alternatives.

Sharing. A downside of implementing a deeply-embedded language via staging is that it is easy to lose sharing introduced by the user in the form of let-bindings and similar constructs. For example, if the user writes:

$$\textbf{let } x = \mathit{expensive} \textbf{ in } f\ x + g\ x$$

then tracing and staging this program as described above will reference the AST produced by the expression '*expensive*' at least twice.[35] This is not what the user intended by writing the let-binding. Approaches exist, in GHC Haskell, to automatically detect and recover sharing of values between multiple positions in a data structure [Gill 2009; McDonell et al. 2013], but these are non-trivial to implement.[36] Furthermore, while a standard common-subexpression elimination (CSE) pass in a compiler might recover the sharing as well, such as pass would be very slow due to having to analyse the full exponentially-sized unfolded AST. In `horde-ad`, at least for the time being, we instead choose the simpler alternative

[34] The 'r' is for *ranked*, meaning that array ranks are reflected on the type level; the library also has a version of the array language for shape-typed arrays (with full shapes on the type-level) as well as a mixed variant.

[35] Assuming that f and g actually use their argument; more than twice if they use their argument multiple times.

[36] Personal experience of the author of this thesis with the Accelerate compiler is that it also becomes fragile when processing many source files in parallel.

of mirroring the solution for conditionals described above. We offer a combinator 'tlet' using which the example can be expressed as follows:

$$\text{tlet } \mathit{expensive} \$ \setminus x \to f\ x + g\ x$$

Because this combinator is implemented by `horde-ad`, its (explicit) sharing can be retained throughout the compilation pipeline. See Section 4.10.3.2 for a more detailed discussion of sharing in the context of the type-class system of `horde-ad`.

4.10.3 Algebra interpretation

The grammar of our core language in Fig. 4.7 not only informs the structure of an AST to *represent* terms of this language at runtime, it also specifies the language itself, the one that the library user writes programs in: independent from any representation, our "core language" consists of a number of syntactic constructs (the ones in Fig. 4.7) together with a semantics for those constructs (namely, their standard interpretation as array operations). However, having just one semantics is sometimes quite limiting: one might want to evaluate programs written in the same syntax using a *different* semantics, for example to compute certain program analyses or to perform partial evaluation. Furthermore, as we will see in Section 4.10.4, the AD that we did in Sections 4.1 and 4.5 can also be seen as an alternative semantics for our syntax, as can indeed ASTs themselves: the latter is how we address the repeated re-differentiation problem identified at the beginning of Section 4.6.

Mathematically, the language of array operations set out in Fig. 4.7 induces a family of *algebras*:[37] each such algebra is a semantics of the array language on some *carrier* data type. We can encode this family of algebras in Haskell using a type class:[38]

```
data Sh k where
  SZ   :: Sh 0
  (:$:) :: Int → Sh k → Sh (1 + k)

class BaseTensor t where    — not the final version! See below.
  tconcrete :: Array k τ → t k τ
  tlet      :: t k σ → (t k σ → t k τ) → t k τ
  tindex    :: t k τ → Ix k → t k τ
  tgather   :: Sh (m + k) → t (n + k) τ →
                          (Ix m → Ix n) → t (m + k) τ
  — ... other methods ...
```

[37] Correctly: a category of F-algebras, where the functor F is induced by the syntax in Fig. 4.7.

[38] The careful reader may note that it is unclear what the expected sharing behaviour of the methods of this type class is. We will clarify the (somewhat subtle) situation after having introduced the basic instances.

The data type Sh, encoding shapes, is analogous to the Ix data type for indices defined in Section 4.5. Regarding the type of 'tlet': in order to be independent of the particular representation of variables in the various semantics for our language, we use higher-order abstract syntax (HOAS) style to encode let-bindings: '**let** $x = s$ **in** t' corresponds to 'tlet s $(\lambda x.\ t)$'. Consequently, there is no "tvar" method in BaseTensor.

The idea is that each algebra in the family is an *instance* of this BaseTensor type class for the appropriate carrier data type. The implementation of the methods for the instance shows in what way the carrier indeed forms an algebra for our language. In other words: the type class is the interface that every proposed carrier must implement in order to be a semantics for our language. For example, there would be an instance of BaseTensor for Array, yielding standard evaluation semantics (a little functional array language); for more details, see below.

The type class may look fine for this purpose at first glance, but to make the plan actually work out, we have to change two things:

1. The data type Ix was originally defined as follows in Section 4.5:

$$\begin{aligned}&\textbf{data}\ \mathrm{Ix}\ k\ \textbf{where}\\ &\quad \mathrm{IZ}\ \ :: \mathrm{Ix}\ 0\\ &\quad (:::) :: \mathrm{Int} \to \mathrm{Ix}\ k \to \mathrm{Ix}\ (k+1)\end{aligned}$$

   but for use in the type class, we must generalise this. The reason is that different semantics (such as *symbolic* array computations, i.e. ASTs, as we will see below) have different ideas about what the 'Int' inside Ix should be. (Indeed, in a symbolic array computation, indices are also symbolic.) As a solution, we let the index components be rank-zero tensors:

$$\begin{aligned}&\textbf{data}\ \mathrm{Ix}\ t\ k\ \textbf{where}\\ &\quad \mathrm{IZ}\ \ :: \mathrm{Ix}\ t\ 0\\ &\quad (:::) :: t\ 0\ \mathrm{Int} \to \mathrm{Ix}\ t\ k \to \mathrm{Ix}\ t\ (1+k)\end{aligned}$$

   Thus if $t = \mathrm{Array}$, this definition is morally the same as the original, as an Array 0 Int is equivalent to a single Int.

2. It turns out that sharing using 'tlet' is insufficient if we want to write dual-numbers reverse AD as an instance of BaseTensor, i.e. as a semantics of our array language. Indeed, this is expected, as the dual-numbers instance of BaseTensor will automatically perform the Delta extraction of Section 4.6, and to do that we needed to introduce share to the core language. Thus, we do so here too:

$$\begin{aligned}&\textbf{class}\ \mathrm{BaseTensor}\ t\ \textbf{where}\\ &\quad \mathrm{tshare} :: t\ k\ \tau \to t\ k\ \tau\\ &\quad \textit{— other methods ...}\end{aligned}$$

The intended meaning of 'tshare' is to produce a tensor that can be duplicated without causing any recomputation. The required ID is generated inside 'tshare'. We will look at methods of sharing again after we have defined some basic instances of BaseTensor.

After these two modifications, we get the following, suitable class formulation:

```
class BaseTensor t where
  tconcrete :: Array k τ → t k τ
  tlet      :: t k σ → (t k σ → t k τ) → t k τ
  tshare    :: t k τ → t k τ
  tindex    :: t k τ → Ix t k → t k τ
  tgather   :: Sh (m + k) → t (n + k) τ → (Ix t m → Ix t n) → t (m + k) τ
  — ... other methods ...
```

4.10.3.1 Basic instances

A natural instance of the BaseTensor type class is the standard evaluation semantics of our language. Its carrier is the Array data type (in `horde-ad` named Concrete to underscore that these are normal physical arrays, not containing any symbolic components), and the interpretations of the language constructs are the usual call-by-value ones on concrete arrays:

```
instance BaseTensor Array where
  tconcrete a    = a
  tlet a f       = f a      — the metalanguage handles sharing.
  tshare a       = a        — ditto.
  tindex a i     = index a i
  tgather sh a f = gather sh a f
  — etc.
```

assuming suitable methods 'index', 'gather', etc. on arrays.

Now, assume we have an AST representation for our language, for example using the following generalised algebraic data type (GADT):

```
data AST k τ where
  Concrete :: Array k τ → AST k τ
  Var      :: VarName k τ → AST k τ
  Let      :: VarName k σ → AST k σ → AST k τ → AST k τ
  Share    :: ID_AST k τ → AST k τ → AST k τ
  Index    :: AST k τ → Ix k → AST k τ
  Gather   :: Sh (m + k) → AST (n + k) τ → (Ix AST m → Ix AST n)
           → AST (m + k) τ
  — etc.
```

For ASTs, we choose normal abstract syntax, as opposed to HOAS, to simplify handling in the below.[39] However, we do need a `Share` constructor to be able to implement the 'tshare' method of BaseTensor. In contrast to our earlier presentation of Delta, where DVarName and ID are indexed just by the rank of the array they represent (because, for simplicity, the element type is always $\mathbb{R}$), VarName and ID_{AST} are indexed by both the rank and the element type.

This AST data type can also be the carrier of a semantics:

instance BaseTensor AST **where**
  tconcrete a = `Concrete` a
  tlet a f = **let** v = VarName GenID **in** `Let` v a (f (`Var` v))
  tshare a = `Share` (ID_{AST} GenID) a
  tindex a i = `Index` a i
  tgather sh a f = `Gather` sh a f
  — *etc.*

In this semantics, the "meaning" of a program is not its evaluation, but instead it is simply its AST: a program *evaluates to* its AST. This instance can be used to implement *staging* in embedded languages: a shallowly embedded language (equivalently, a language in tagless-final style [Carette et al. 2009]) is an embedded language where the programmer interface is essentially a type class like our BaseTensor. If such an embedded language implementation wants to do e.g. a whole-program transformation on the embedded program, it can use an instance of the type class for an AST data type to get an inspectable representation of the program. This approach is called *staging* because there are now two stages of evaluation: the user program evaluates to an AST, and this AST is later (presumably) evaluated itself to some final result.

We have already discussed some design decisions and limitations of the staging implementation in `horde-ad` above in Section 4.10.2. The general framework of this implementations is as described here — via Haskell type-classes. In the next subsection we explain the interplay of sharing, a particularly subtle point of staging, with the algebra interpretation (type-classes) approach.

4.10.3.2 Sharing

For technical reasons as well as to guard the efficiency of the full algorithm that this chapter describes, the sharing-related semantics of the BaseTensor methods is a bit subtle. Let us make clear what is going on.

- 'tlet' models a let-binding with lexical scoping. That is to say: the expression 'tlet a f' is *semantically* equivalent to f a, but 'tlet' ensures that the resulting

[39] An alternative approach is to use PHOAS [Chlipala 2008].

tensor (i.e. the result of 'tlet $a\ f$') does not involve, or represent, multiple redundant computations of a. In the case of the instance for Array, this is moot: assuming that the metalanguage has reference-passing semantics (which is true for Haskell), 'tlet $a\ f = f\ a$' fulfills this goal perfectly.

However, for the instance for AST, this is quite important to get right. Consider the following (contrived) function written against the BaseTensor interface, taking an argument array a of length n:

$$
\begin{aligned}
&\mathit{foo1} :: \text{BaseTensor}\ t \Rightarrow t\ 1\ \mathbb{R} \to t\ 0\ \mathbb{R}\\
&\mathit{foo1}\ a = \text{tlet}\ (\text{tgather}\ (n :\$: \text{SZ})\ a\ (\lambda(i ::: \text{IZ}).\ (n - 1 - i) ::: \text{IZ}))\\
&\qquad\qquad (\lambda a'.\ \text{tindex}\ a'\ (0 ::: \text{IZ}) +_{\mathbb{R}} \text{tindex}\ a'\ (1 ::: \text{IZ}) +_{\mathbb{R}}\\
&\qquad\qquad\qquad \text{tindex}\ a'\ (2 ::: \text{IZ}))
\end{aligned}
$$

('$+_{\mathbb{R}}$' is one of the methods of BaseTensor that we elide in the code snippets in this section to save space and to prevent tedious repetition. Its type is $(+_{\mathbb{R}}) :: t\ k\ \mathbb{R} \to t\ k\ \mathbb{R} \to t\ k\ \mathbb{R}$ and it is one of the binary *op*s in the grammar in Fig. 4.7.) In *foo1*, the 'tgather' computes the reverse of a, after which we take the sum of the first three elements of that computed reverse. If we instantiate t to AST, then it is quite important that 'tlet $a\ f$' is not simply '$f\ a$', but actually creates a `Let` node! Otherwise the produced AST will, when evaluated, recompute the reverse of the input array three times.

- 'tshare' models *global sharing*, as used in Sections 4.1.5 and 4.6.3. Analogous to Share in Delta, the intended meaning of 'tshare' is that if the tensor that it returns is used in multiple places (probably as arguments to other BaseTensor methods of the same instance), this does not lead to duplicate computation of the tensor wrapped by 'tshare'. The same example could be written as follows using tshare instead:

$$
\begin{aligned}
&\mathit{foo2} :: \text{BaseTensor}\ t \Rightarrow t\ 1\ \mathbb{R} \to t\ 0\ \mathbb{R}\\
&\mathit{foo2}\ a =\\
&\quad \textbf{let}\ a' = \text{tshare}\ (\text{tgather}\ (n :\$: \text{SZ})\ a\ (\lambda(i ::: \text{IZ}).\ (n - 1 - i) ::: \text{IZ}))\\
&\quad \textbf{in}\ \text{tindex}\ a'\ (0 ::: \text{IZ}) +_{\mathbb{R}} \text{tindex}\ a'\ (1 ::: \text{IZ}) +_{\mathbb{R}} \text{tindex}\ a'\ (2 ::: \text{IZ})
\end{aligned}
$$

When instantiated to the AST instance, the subterm corresponding to the meta-variable a' will indeed occur three times in the resulting AST, but because all three occurrences are wrapped in a `Share` constructor containing the same ID, an evaluator will memoise the computed value (the reversed array) and not recompute it the second and third time it encounters this same `Share` node. This is analogous to how Share nodes in a Delta term were handled in *reversePass*, except without the requirement that these globally shared subterms are computed in any particular order.

- The other methods of the type class make *no guarantees*, in general, about the sharing that they preserve. That is to say: the 'tlet' in *foo1* and the 'tshare' in *foo2* are necessary, because $(+_{\mathbb{R}})$ and 'tindex' may cheerfully assume their arguments are used only once. But not all instances will duplicate work: of course, if one somehow knows that *foo1* is only going to be instantiated to the Array instance of BaseTensor, a meta-language (Haskell) **let** expression suffices — indeed, 'tlet' and 'tshare' for Array do not do anything more than that.

Later, when we define more instances of BaseTensor, we will refer back to this and explain how those instances are consistent with these rules.

The *interpret* function described below in Section 4.10.3.3 (that interprets an AST into some other instance of the BaseTensor class) can not easily support let-style sharing and global sharing (Share-style sharing, denoted in `horde-ad` by 'tshare') in the same term. The reason is the fact that the BaseTensor methods (see below) are written in a higher-order fashion; not only 'tlet' is, but also things like 'tgather'. A proper interpreter that handles global sharing correctly has to thread a memoisation map through the program, containing the evaluated result for every $\mathrm{ID}_{\mathrm{AST}}$ it encountered inside a share-node. But given the type signature of 'tlet', the interpreter has no way to export the IDs it memoised inside the *body* of the let, to outside that let!

It would be possible to fix 'tlet' to allow a value to be returned from the body in the meta-language:

$$\mathrm{tlet}' :: t\ k\ \sigma \to (t\ k\ \sigma \to (a, t\ k\ \tau)) \to (a, t\ k\ \tau)$$

allowing *interpret* to get the output memoisation map from the body of the created let-binding. However, this same trick cannot be applied to 'tgather' and similar: unlike 'tlet', the argument to 'tgather' might be called only once (for the AST instance of BaseTensor) or many times (for the Array instance of BaseTensor), so neither returning an extra a from the hypothetical tgather′, nor an extra Array $m\ a$, would always work.

Fortunately, in our algorithm, we do not need to handle the fully-general case: in terms that we need to interpret, we never have an actual mix of lets and global sharing. More precisely, in this chapter we never put 'tshare' inside the body of a 'tlet'. This allows us to:

1. Implement a conversion from global sharing to let-based sharing on ASTs that can consider `Let` nodes black-boxes, and handle `Share` nodes only. This is the *shareToLet* function of Section 4.6.4.

2. Express *interpret* on let-bindings only, with no support for `Share`.

This observation of no-share-inside-let makes the semantics of our terms clear and *interpret* feasible.

In the `horde-ad` library implementation, the invariant of no 'tshare' inside 'tlet' is enforced by the typing of the grammar. Moreover, the typing ensures that *interpret* is only ever called on terms with no global sharing in them at all.

4.10.3.3 Interpretation

One of the reasons for having the BaseTensor type class is that one can write array computations polymorphic in the specific tensor type, and later instantiate them to multiple backends, or assign them different semantics by instantiating them to non-standard instances. However, we have already run the user program through the BOT in Section 4.4, so we have an AST now, not a polymorphic function. Fortunately, this is no obstacle, because the AST instance of BaseTensor is somewhat special: its values (ASTs)[40] can be uniquely[41] interpreted into any other semantics of the language. The type of this interpretation function is:

$$\mathit{interpret} :: \text{BaseTensor } t \Rightarrow \text{DMap}_2 \text{ VarName } t \to \text{AST } k\ \tau \to t\ k\ \tau$$

The first parameter (the DMap_2) is the environment giving the interpretation for any free variables (`Var`) that occur in input term. We define DMap_2 in terms of DMap by uncurrying f and g:

$$\text{DMap}_2\ f\ g = \text{DMap}\ (\lambda(k, \tau).\ f\ k\ \tau)\ (\lambda(k, \tau).\ g\ k\ \tau)$$

and then defining methods on DMap_2 analogous to the ones on DMap. For example, it is instructive to look at the types of the two versions of lookup (where DMap.lookup is from Fig. 4.17):

$$\begin{aligned}&\text{DMap.lookup} &&:: \texttt{GCompare}\ f \Rightarrow f\ a \to \text{DMap}\ f\ g \to \text{Maybe}\ (g\ a)\\ &\text{DMap}_2\text{.lookup} &&:: \texttt{GCompare}_2\ f \Rightarrow f\ a\ b \to \text{DMap}_2\ f\ g \to \text{Maybe}\ (g\ a\ b)\end{aligned}$$

That is to say: DMap_2 is to data types with 2 type parameters what DMap is to data types with 1 type parameter.

As an example usage of *interpret*, specialising the type variable t to 'Array' and passing an empty initial enrivonment (DMap_2.empty), one obtains:

$$\mathit{interpret}' :: \text{AST } k\ \tau \to \text{Array } k\ \tau$$

which evaluates a closed term to its value as an array.

[40] With the let/share sharing caveats already mentioned.

[41] Assuming we want our interpretation to commute with primitives in our language, which seems quite reasonable. This requirement comes from universal algebra: the AST algebra is a term algebra and thereby an *initial* algebra in our family.

Despite implementing a fairly fundamental function, *interpret* is somewhat cumbersome to write. The definition we use is given in Fig. 4.23; let us walk through its major components.

The clauses of *interpret* itself map each of the AST constructors (i.e. primitives in our language) to the corresponding BaseTensor method on t; the environment is extended as needed for let-bindings, and as discussed, `Share` is unsupported. Aside from subterms, ASTs also contain more complicated structures such as index values (in e.g. `Index`) and index mapping functions (in e.g. `Gather`). An index (interpreted by *interIx*) is just a list of terms. To interpret an index mapping function (*interIxFun*), we implement this diagram:

$$\begin{array}{ccc} \text{Ix AST } m & \xrightarrow{\quad f \quad} & \text{Ix AST } n \\ \uparrow & & \downarrow \\ \textit{extend} & \xrightarrow{\quad env \quad} & \textit{interIx} \\ \downarrow & & \downarrow \\ \text{Ix } t\ m & & \text{Ix } t\ n \end{array}$$

That is to say: we generate variable names for the components of the input index, pass the resulting symbolic index through the symbolic mapping function, and then compute the output index by evaluating the output symbolic index with the generated variable names mapped to the components of the input index.

The GenID operation is an effect that generates a unique ID impurely. This is possible in Haskell using an `IORef` with `unsafePerformIO` and the correct tricks. While this breaks referential transparency of our ostensibly pure algorithm, it only does so in a controlled way; in particular, the semantics of our terms is invariant under bijective transformation of the assigned IDs (preserving their order, in the case of ID values in Delta), so e.g. the starting ID does not matter.

4.10.4 AD as an algebra interpretation

It turns out that because of the nice, compositional nature of the AD code transformation that produces the forward pass (Figs. 4.2, 4.15 and 4.16), it can be written as an algebra interpretation on our core language. The carrier data type here is a dual array: a pair of an array and a Delta term.

```
data ADVal k τ = ADVal (Array k τ) (Delta k)
instance BaseTensor ADVal where
  tconcrete a              = ADVal a Zero
  tlet (ADVal p d) f       = f (ADVal p d)
  tshare (ADVal p d)       = ADVal p d
  tindex (ADVal p d) ix    = ADVal (tindex p ix) (Share GenID (Index d ix))
  tgather sh (ADVal p d) f = ADVal (tgather sh p f)
                                   (Share GenID (Gather d f))
  — etc.
```

```
interpret :: BaseTensor t ⇒ DMap_2 VarName t → AST k τ → t k τ
interpret env (Concrete t)     = tconcrete t
interpret env (Var v)          = case DMap_2.lookup v env of
                                   Just x → x
                                   Nothing → error "Free variable"
interpret env (Let v s t)      = tlet (interpret env s)
                                      (λx. interpret (DMap_2.insert v x env) t)
interpret env (Share id t)     = error "Unimplemented"
interpret env (Index t i)      = tindex (interpret env a) (interIx env i)
interpret env (Gather sh a f) = tgather sh (interpret env a) (interIxFun env f)
— etc.

interIx :: BaseTensor t ⇒ DMap_2 VarName t → Ix AST k → Ix t k
interIx env IZ       = IZ
interIx env (i ::: ix) = interpret env i ::: interIx env ix

interIxFun :: BaseTensor t ⇒ DMap_2 VarName t
           → (Ix AST m → Ix AST n) → (Ix t m → Ix t n)
interIxFun env f ix = let (env', ix') = extend env ix in interIx env' (f ix')
    where extend :: DMap_2 VarName t → Ix t k
                 → (DMap_2 VarName t, Ix AST k)
          extend env IZ       = env
          extend env (i ::: ix) = let (env', ix') = extend env ix
                                      v = VarName GenID
                                  in (DMap_2.insert v i env', Var v ::: ix')
```

Figure 4.23: Interpreter from AST to an arbitrary instance of BaseTensor. In other words: this implements the unique homomorphism (function that commutes with all the primitives in our language) from the initial algebra to another algebra on the same language. While the function is unique, its *implementation* is not, but the one given here has the advantage of being parametrically polymorphic over all BaseTensor instances.

The implementations of the BaseTensor methods for ADVal are adapted directly from the equations of the D code transformation from Section 4.5 (Figs. 4.15 and 4.16), with the (meta-)pairs used there replaced by uses of the ADVal constructor. The implementations of 'tlet' and 'tshare', however, require some justification. As before in the Array instance, the sharing operations (tlet and tshare) have no effect on the primal (left) half because Haskell does not recompute values when you use them multiple times. For the dual (right) half, however, we can also ignore the sharing operations: because the Delta terms created by the other methods in the instance are always wrapped inside a Share node and thus freely duplicable in the meta-language, d in the argument to 'tlet' or 'tshare' will be freely duplicable. Hence, wrapping it in another Share node does not achieve anything, and we choose to omit the redundant wrapper for efficiency.

Having the carrier be a *pair* of two tensor-like things (an array and a Delta term, in this case) means that any computation that is interpreted into this algebra gets repeated twice: once on arrays and once on Delta terms. Furthermore, while the Delta terms may refer to the primal half of the computation, there is no dependency the other way round. Thus, when a BaseTensor-polymorphic function, for example:

$$dotprod :: \text{BaseTensor } t \Rightarrow t\ 1\ \mathbb{R} \to t\ 1\ \mathbb{R} \to t\ 0\ \mathbb{R}$$

is interpreted in the dual-numbers algebra ADVal, its result *is a pair* of a primal result and a Delta term that by construction are fully separated. This is good, because it means that we no longer have the entangling of the primal and dual halves of the AD output that we started out with at the beginning of Section 4.6, without having to do any manual simplification like in Section 4.6.6: the simplification is done for us by the embedding into the meta-language.

4.10.4.1 Generalisation

Looking at the BaseTensor instance for ADVal, we notice that the primal halves of the returned pairs simply mirror the BaseTensor methods they are implementing: 'tindex' maps to 'tindex', etc., as expected. After all, the primal computation of a derivative program performs the same computations on scalars and arrays as the original program did. This means that we can generalise ADVal: its primal component need not be an Array, and could instead be any BaseTensor type. Thus, as a first attempt we can try to simply parametrise ADVal on the tensor algebra used for the primal operations like this:

$$\textbf{data}\ \text{ADVal}'\ t\ k\ a = \text{ADVal}\ (t\ k\ a)\ (\text{Delta}\ k)$$

but this does not quite work. The reason is that primal tensors end up in the Delta term as well. Consider the rule for multiplication: (compare the original

code transformation for scalars in Fig. 4.3)

```
instance BaseTensor ADVal where
  ADVal p1 d1 ×ℝ ADVal p2 d2 =
    ADVal (p1 ×ℝ p2) (Share GenID (Add (Scale p2 d1) (Scale p1 d2)))
```

The Scale constructor of Delta contains a tensor from the primal half of the dual-numbers pair, so if we generalise the primal tensor type, we must generalise the tensor type in Scale as well. Let us do so:

```
data Delta t k where
  — ... other constructors ...
  Scale :: t k ℝ → Delta t k → Delta t k
  — ... other constructors ...

data ADVal t k a = ADVal (t k a) (Delta t k)
```

Now it becomes straightforward to lift the previous BaseTensor instance for ADVal to the generalised, parametrised ADVal; we just need to take care to use 'tconcrete' explicitly for injecting constants into the primal computation, and to use explicit 'tshare' on primal terms when duplicating them:

```
instance BaseTensor t ⇒ BaseTensor (ADVal t) where
  tconcrete a                = ADVal (tconcrete a) Zero
  tlet (ADVal p d) f         = f (ADVal (tshare p) d)
  tshare (ADVal p d)         = ADVal (tshare p) d
  tindex (ADVal p d) ix      = ADVal (tindex p ix) (Share GenID (Index d ix))
  tgather sh (ADVal p d) f = ADVal (tgather sh p f)
                                   (Share GenID (Gather d f))
  — ... other methods ...
```

Most of the code remains unchanged; the result is a parametrised algebra interpretation[42] into dual arrays.

Notable in this BaseTensor instance is that let-bindings are interpreted into global sharing. The reason for this choice is quite subtle: because primal tensors end up inside Delta terms, and Delta terms are not lexically-scoped subterms of our primal computation (after all, we want to *disentangle* the two, not interleave both in the same computation!), we cannot use 'tlet' to interpret the sharing in the primal computation. With global sharing, on the other hand, we can choose to scope the namespace of $\mathrm{ID}_{\mathrm{AST}}$s of globally shared primal values over the whole computation, not just the primal half; that way, the dual computation depends on

[42] Formally, this defines a homomorphism between the algebras t and ADVal t, and is an example of a derived algebra morphism.

the primal computation, but not the other way round. Hence, the two are still disentangled.

In effect, we thus create a single namespace of IDs for primal tensors ($\mathrm{ID}_{\mathrm{AST}}$) over all primal and dual values in an ADVal computation, and a separate namespace (ID) just within the Delta terms. The primal tensor IDs are referenced in the primal computation as well as in the embedded tensor values inside Scale in Delta, using whatever method the tensor type t uses to record global sharing; the IDs referring to Delta terms are just referenced using Share constructors. These namespaces are disjoint, not only because of the differing ID types, but also because the former encodes sharing of primal tensors and the latter encodes sharing of Delta terms, which are different types.

4.10.4.2 Instantiation

At this time we can finally solve, using the type class implementation, the problem of repeated re-differentiation that we approached using code transformations in Section 4.6. The trick is that we can instantiate this parametrised algebra interpretation (ADVal) to ASTs, yielding 'ADVal AST k a': a pair of an AST and a Delta term containing ASTs.

$$\text{ADVal AST } k\ a \approx (\text{AST } k\ a, \text{Delta AST } k)$$

We call these pairs *symbolic dual arrays*. Because we have a BaseTensor instance for AST and a parametrised one for ADVal as shown above, this instantiated type is also an instance of BaseTensor. This means that we can interpret programs into it using our *interpret* function! What does the result look like?

- Where the original program took arrays as input, the reinterpreted program takes symbolic dual arrays as input. In particular, the reinterpreted program can be run to completion (symbolically) without supplying concrete input arguments: free-variable AST nodes suffice.

- Because of the simple design of our core language, a program returns exactly one tensor as output. Thus, the program output will be one symbolic dual array.

- The primal half of this dual array is an AST that, when evaluated (i.e. interpreted into a concrete array algebra) computes the original value of the program. Note that this AST uses global sharing, so before it can be interpreted, the global sharing must be transformed into local sharing ('tshare' into 'tlet') using the conversion in Section 4.6.4.

- The dual half of the output dual array is a Delta term containing ASTs (that reference values computed in the primal half): this describes the (symbolic) forward derivative of the program evaluated at the given (symbolic) inputs.

In Section 4.5, evaluation of a Delta term proceeded by passing it to *reversePass*, which took an incoming cotangent and a (non-symbolic) Delta term and produced a sparse gradient, which could be materialised into a full gradient in the wrapper around the algorithm. Surprisingly, the operations that *eval*, and hence *reversePass*, performs on the cotangents are precisely those that comprise the core language: this works because we have designed our core language to be closed under differentiation (assuming sufficient primitive arithmetic operators). For example, we have not only gather but also scatter, and not only sumOuter but also replicate.

Hence, we can generalise *reversePass* to work on arbitrary tensor algebras. Doing this, we end up with the following types:

$$
\begin{aligned}
&\textbf{data}\ \mathrm{ES}\ t = \mathrm{ES} \\
&\quad \{\ \mathrm{grad} :: \mathrm{DMap}\ \mathrm{DVarName}\ (\lambda k.\ t\ k\ \mathbb{R}) \\
&\quad ,\ \mathrm{dfrag} :: \mathrm{DMap}\ \mathrm{ID}\ (\mathrm{Delta}\ t) \\
&\quad ,\ \mathrm{accum} :: \mathrm{DMap}\ \mathrm{ID}\ (\lambda k.\ t\ k\ \mathbb{R})\ \} \\
&\mathit{reversePass} :: \mathrm{BaseTensor}\ t \Rightarrow t\ k\ \mathbb{R} \to \mathrm{Delta}\ t\ k \\
&\qquad\qquad \to \mathrm{DMap}\ \mathrm{DVarName}\ (\lambda k.\ t\ k\ \mathbb{R}) \\
&\mathit{eval} \qquad :: \mathrm{BaseTensor}\ t \Rightarrow t\ k\ \mathbb{R} \to \mathrm{Delta}\ t\ k \to \mathrm{ES}\ t \to \mathrm{ES}\ t \\
&\mathit{backprop} \quad :: \mathrm{BaseTensor}\ t \Rightarrow \mathrm{ES}\ t \to \mathrm{ES}\ t
\end{aligned}
$$

with basically identical implementations to those given in Section 4.5.

By this point, we have a fairly complete implementation of reverse AD for our language, designed and implemented in a compositional, modular manner. In particular, as a result of the compositionality of the design:

- If we remove BOT, the AD algorithm can be written directly as a *shallow embedding*[43], of course losing efficient differentiation of array indexing, but gaining expressiveness of the source language (more expressive dynamic control flow) because there is no staging any more: all control flow is traced away.

- The AD algorithm is completely decoupled from BOT, the interface being purely an AST of the core language (Fig. 4.7).

What is missing for a comprehensive picture is an overview of the full pipeline, and the wrapper around the algorithm that makes it usable. These are not too

[43]This is also known as a *final* encoding of the algorithm, as opposed to an *initial* encoding which goes via an initial algebra, i.e. an algebraic data type (the AST).

hard to derive by generalizing the pipelines and wrappers from the previous sections and their implementation can be inspected in the `horde-ad` source code.

5 The CHAD Algorithm

CHAD (Combinatory Homomorphic Automatic Differentiation) is a define-then-run code transformation for automatic differentiation first described in [Vákár and Smeding 2022], and builds on similar ideas of Vytiniotis et al. [2019] and Elliott [2018]. The transformation is compositional: the derivative of a term is defined directly in terms of the derivatives of its subterms. The transformation is also very strongly typed: after fixing the derivatives of the primitive operations in the source language, the typing restricts the transformation for most basic constructs of an extended lambda calculus (pairing, let-binding, case-distinction, etc.) to the point that there is only one reasonable implementation. To be "reasonable", it is typically sufficient to ensure that all available values are used exactly once.[1] Correctness of the transformation follows using a logical relations argument. [Vákár and Smeding 2022]

In a theoretical line of work, the semantical theory of the algorithm is extended to coproducts and (co)inductive types by Nunes and Vákár [2023] and to iteration by Nunes, Plotkin, and Vákár [2025]. This thesis complements this research line with a perspective on efficiency: in Chapter 6 (published as [Smeding and Vákár 2024]) we analyse and improve the operational behaviour of output programs of CHAD by addressing time complexity issues; in Section 6.7 and Chapter 7 we consider, and significantly improve, practical (parallel) efficiency.

First, however, the current chapter introduces the (inefficient) base algorithm defined in [Vákár and Smeding 2022] (plus coproducts from [Nunes and Vákár 2023]) for a functional programming audience and gives some intuition for why it is correct, *without* reference to the category theory from which the definitions are derived: readers interested in the mathematics are referred to the cited articles. We will use some terminology and notation introduced in the Background chapter, Section 2.2.

This chapter is freshly written for this thesis and represents [Vákár and Smeding 2022].

[1] Mathematically, CHAD implements the unique structure-preserving (i.e. preserving products, coproducts and exponentials) functor from the (freely generated syntactic category of the) source language to a specially constructed target syntactic category. For details, see [Nunes and Vákár 2023].

$$
\begin{array}{lll}
\sigma, \tau & ::= & \mathbb{R} \mid \mathbb{Z} \mid \mathbf{1} \mid \sigma \times \tau \mid \sigma \sqcup \tau \mid \sigma \to \tau \\
s, t & ::= & x \mid \mathbf{let}\ x : \tau = s\ \mathbf{in}\ t \qquad \text{(variables)} \\
& \mid & \langle\rangle \mid \langle s, t\rangle \mid \mathrm{fst}\ t \mid \mathrm{snd}\ t \qquad \text{(unit and products)} \\
& \mid & \mathrm{inl}\ t \mid \mathrm{inr}\ t \qquad \text{(coproducts)} \\
& \mid & \mathbf{case}\ s\ \mathbf{of}\ \{\ \mathrm{inl}\ x \to t_1 \mid \mathrm{inr}\ y \to t_2\ \} \qquad \text{(coproducts)} \\
& \mid & \lambda x.\ t \mid s\ t \qquad \text{(functions)} \\
& \mid & r \mid \mathrm{sign}\ t \mid \mathrm{op}_{\mathbb{R}}(t_1, \ldots, t_n) \qquad \text{(real scalars, op} \in \mathrm{Op}_n^{\mathbb{R}}\text{)} \\
& \mid & n \mid \mathrm{op}_{\mathbb{Z}}(t_1, \ldots, t_n) \qquad \text{(integers, op} \in \mathrm{Op}_n^{\mathbb{Z}}\text{)}
\end{array}
$$

Figure 5.1: The source language for naive CHAD.

5.1 The transformation

As in the first half of this thesis, we focus on reverse AD; the reasoning (and mathematics) for forward AD is similar but results in an algorithm that is for most purposes worse than dual-numbers forward AD. In this section and its subsections, we first derive the (reverse AD) transformation on types step-by-step to its full generality; the reader will see that the types indeed leave few options for a sensible implementation of the term transformation. Afterwards, we present the full code transformation in Section 5.1.5. Adventurous readers may prefer to skip ahead to Section 5.1.5 after seeing the language definition, referring back for design motivation if necessary.

Source language. A grammar for the types and terms of the source language for this chapter is shown in Fig. 5.1. The types of this language are real scalars ($\mathbb{R}$), integers ($\mathbb{Z}$), units ($\mathbf{1}$), product types ($\sigma \times \tau$), coproducts / sum types ($\sigma \sqcup \tau$) and function types ($\sigma \to \tau$). Most of the term syntax elements should be familiar and have the standard typing rules. The intended semantics is call-by-value, and let-bindings are non-recursive. The terms for dealing with scalars and integers have the following meaning:

- r stands for a scalar constant such as 6.28; n stands for an integer constant.
- We have sets $\mathrm{Op}_n^{\mathbb{R}}$ of n-ary scalar primitive operations and $\mathrm{Op}_n^{\mathbb{Z}}$ of n-ary integer primitive operations. An $\mathrm{op}_{\mathbb{R}} \in \mathrm{Op}_n^{\mathbb{R}}$ (analogously $\mathrm{op}_{\mathbb{Z}} \in \mathrm{Op}_n^{\mathbb{Z}}$) represents a possibly partial function $\mathbb{R}^n \rightharpoonup \mathbb{R}$; primitive operations are always fully applied and written $\mathrm{op}_{\mathbb{R}}(t_1, \ldots, t_n)$ when applied to arguments $t_1, \ldots, t_n$. For example, $\sin \in \mathrm{Op}_1^{\mathbb{R}}$; $(+_{\mathbb{R}}), (\cdot_{\mathbb{R}}) \in \mathrm{Op}_2^{\mathbb{R}}$; and $(+_{\mathbb{Z}}), \mathrm{mod} \in \mathrm{Op}_2^{\mathbb{Z}}$. As usual in this thesis, we model these operations abstractly to show the general structure for supporting primitive operations in the AD algorithm.

- 'sign' is the signnum function of type $\mathbb{R} \to \mathbf{1} \sqcup \mathbf{1}$, using $\mathbf{1} \sqcup \mathbf{1}$ to model booleans. sign $(-2) = \text{inl}\ \langle\rangle$; sign $3 = \text{inr}\ \langle\rangle$; what sign 0 evaluates to, we leave to the reader. This operation 'sign' is a stand-in for any non-trivial condition on real scalars, and is intentionally simplistic. In a useful language, one would naturally also include $(<)$, $(\geq)$, etc.; such additional operations interact with differentiation exactly like 'sign' does. We could handle general discrete operations abstractly in the algorithm too, but $\text{op}_{\mathbb{R}}$ and $\text{op}_{\mathbb{Z}}$ already show the patterns, and doing it here again would introduce some additional typing that obscures rather than elucidates the point.

The language includes basic dynamic control flow (conditionals) in the form of **case**, which CHAD is able to differentiate to proper dynamic control flow; this is in contrast to the algorithm in Chapter 4, where the BOT (Section 4.4) turned most conditionals into strict selections.

On the other hand, the reader may note that this language has neither looping nor recursion constructs. Fortunately, both can be added without issue, although their derivatives are somewhat tricky; the reason we elide them in this chapter is to focus on the core ideas. The theory has been developed for looping (iteration) [Nunes et al. 2025], and in unpublished form for recursion.

Now that we have a language, let us think about how to represent derivatives.

5.1.1 Simply-typed cotangents

The (co)tangent typing of CHAD has a strong connection to differential geometry; for the interested reader, we link CHAD to the mathematical terminology in Section 5.2. However, the typing can also be motivated using some functional programming wisdoms: (1) incompatible values should have different types; and (2) make illegal states unrepresentable.

We start by deriving from the first wisdom that it is beneficial for cotangents to have different typing from primals: they interact in few, controlled ways, and it is easy to mix them up. Hence, let us write[2] $\mathcal{D}[\tau]_2$ for the cotangent type corresponding to a certain primal type τ — that is to say: reverse derivatives for a value $x : \tau$ will be of type $\mathcal{D}[\tau]_2$. Does it work to take $\mathcal{D}[\tau]_2 = \tau$?

Certainly it makes sense that $\mathcal{D}[\mathbb{R}]_2 = \mathbb{R}$ and $\mathcal{D}[\mathbf{1}]_2 = \mathbf{1}$, and even $\mathcal{D}[\sigma \times \tau]_2 = \mathcal{D}[\sigma]_2 \times \mathcal{D}[\tau]_2$. But what of $\mathcal{D}[\mathbb{Z}]_2$? If we have a function $f : \mathbb{Z} \times \mathbb{R} \to \mathbb{R}$, should its gradient contain a partial derivative with respect to the integer argument, and if so, what information should it contain?

CHAD observes that a derivative with respect to an integer input contains no information (nor does a derivative of an integer output with respect to something else), as integers have no continuous structure: any "small increment" to an

[2]The point of the subscript '2' will become clear when we add function types in Section 5.1.4.

integer value would be zero. Hence, CHAD sets $\mathcal{D}[\![\mathbb{Z}]\!]_2 = \mathbf{1}$ (as $\mathbf{1}$ is the type storing no information), resulting in the following types so far:

$$\begin{aligned}
&\mathcal{D}[\![-]\!]_2 : \mathsf{Type} \to \mathsf{Type}\\
&\mathcal{D}[\![\mathbb{R}]\!]_2 = \mathbb{R}\\
&\mathcal{D}[\![\mathbb{Z}]\!]_2 = \mathbf{1}\\
&\mathcal{D}[\![\mathbf{1}]\!]_2 = \mathbf{1}\\
&\mathcal{D}[\![\sigma \times \tau]\!]_2 = \mathcal{D}[\![\sigma]\!]_2 \times \mathcal{D}[\![\tau]\!]_2
\end{aligned}$$

$\mathcal{D}[\![-]\!]_2$ is a normal (type-level) function; we write applications of it specially as $\mathcal{D}[\![\tau]\!]_2$ instead of $\mathcal{D}_2\ \tau$ for consistency with existing literature on CHAD and to increase legibility.

Leaving coproducts and functions aside for a moment, we can now write down the "type" of the CHAD transformation on terms.

5.1.2 Simply-typed transformation

Imagine for a moment that we are not differentiating terms, but instead closed functions $f : \sigma \to \tau$. In reverse AD we would like to produce a function that computes $(Df)^\top\ x\ u$ for any x and u, so the top-level type that we might expect for the reverse derivative f' is $\sigma \to \mathcal{D}[\![\tau]\!]_2 \multimap \mathcal{D}[\![\sigma]\!]_2$: given an input $x : \sigma$, produce a linear function that takes a cotangent to the output ($u : \mathcal{D}[\![\tau]\!]_2$) and returns the corresponding cotangent to the input ($(Df)^\top\ x\ u : \mathcal{D}[\![\sigma]\!]_2$). Note that the linear function arrow we use here ($\multimap$) is just a normal function in an implementation; the linearity annotation is only there on paper and indicates that the function is vector space homomorphism.[3]

However, this type is insufficient. As we have seen in Section 2.2.6, the forward pass of reverse AD also naturally produces the output $f\ x : \tau$; in fact, it turns out that producing this output is essential to make the algorithm compositional using the chain rule of differentiation. Hence, we instead produce a function of type $\sigma \to (\tau \times (\mathcal{D}[\![\tau]\!]_2 \multimap \mathcal{D}[\![\sigma]\!]_2))$.[4]

To translate these ideas to CHAD, which differentiates open terms instead of closed functions, we note that we can regard the free variables of a term t as its inputs; the term then implements a function from those free variables to whatever the term returns. For example, the term t given by:

$$\mathbf{let}\ z = x +_{\mathbb{R}} y\ \mathbf{in}\ z \cdot_{\mathbb{R}} y$$

[3] For CHAD, monoid homomorphisms are in fact sufficient; more discussion on page 210.

[4] The reader may wonder why we do not write $\sigma \times \mathcal{D}[\![\tau]\!]_2 \to \tau \times \mathcal{D}[\![\sigma]\!]_2$ instead, in analogy to dual-numbers forward AD. The answer is that the primal is necessary to determine what the reverse pass should even look like; to see this, we invite the reader to continue reading until Eq. (5.4) and then attempt writing the transformation for let-bindings with this alternative typing.

has two free variables, $x : \mathbb{R}$ and $y : \mathbb{R}$, and returns a single real scalar; we denote this using the typing judgement $x : \mathbb{R}, y : \mathbb{R} \vdash t : \mathbb{R}$. Because t essentially implements a function $\mathbb{R}^2 \to \mathbb{R}$, we expect reverse AD to produce a function of type $\mathbb{R}^2 \to (\mathbb{R} \times (\mathcal{D}[\mathbb{R}]_2 \multimap \mathcal{D}[\mathbb{R}^2]_2))$. Specifically, CHAD produces a term t' satisfying $x : \mathbb{R}, y : \mathbb{R} \vdash t' : \mathbb{R} \times (\mathcal{D}[\mathbb{R}]_2 \multimap \mathcal{D}[\mathbb{R}^2]_2)$.

More generally, CHAD is a code transformation $\mathcal{D}_\Gamma[t]$ working on terms $\Gamma \vdash t : \tau$; the output of the code transformation satisfies:

$$\Gamma \vdash \mathcal{D}_\Gamma[t] : \tau \times (\mathcal{D}[\tau]_2 \multimap \mathcal{D}[\Gamma]_2)$$

where we abuse notation a bit to extend $\mathcal{D}[-]_2$ to work on environments as follows:

$$\mathcal{D}[\varepsilon]_2 = \mathbf{1} \qquad \mathcal{D}[\Gamma, x : \tau]_2 = \mathcal{D}[\Gamma]_2 \times \mathcal{D}[\tau]_2$$

Note that the result of applying $\mathcal{D}[-]_2$ to a typing environment is a type, not another environment.

We can summarise the "type" of CHAD in what may be called the *meta-type* of the transformation (somewhat simplified so far):

$$\Gamma \vdash t : \tau \quad \rightsquigarrow \quad \Gamma \vdash \mathcal{D}_\Gamma[t] : \tau \times (\mathcal{D}[\tau]_2 \multimap \mathcal{D}[\Gamma]_2) \tag{5.1}$$

We write the code transformation explicitly subscripted by the environment Γ of the input term, both to distinguish it notationally from the transformation $\mathcal{D}[-]_2$ on types and to make explicit in what environment we consider the term t to live.

To simplify discussion, we introduce some terminology: fst $\mathcal{D}_\Gamma[t] : \tau$ is called the *primal*, and snd $\mathcal{D}_\Gamma[t] : \mathcal{D}[\tau]_2 \multimap \mathcal{D}[\Gamma]_2$ is called the *backpropagator*. Until we introduce support for function types in Section 5.1.4, the primal will always compute the same result as the original term.

Some rules. With the typing so far, we can already give some of the code transformation rules. As an example, let us look at product introduction:[5]

$$\begin{aligned} \mathcal{D}_\Gamma[\langle s : \sigma, t : \tau \rangle] = \mathbf{let}\ & \langle x : \sigma, dx : \mathcal{D}[\sigma]_2 \multimap \mathcal{D}[\Gamma]_2 \rangle = \mathcal{D}_\Gamma[s] \\ & \langle y : \tau, dy : \mathcal{D}[\tau]_2 \multimap \mathcal{D}[\Gamma]_2 \rangle = \mathcal{D}_\Gamma[t] \\ \mathbf{in}\ & \langle \langle x, y \rangle, \underline{\lambda} \langle d_1, d_2 \rangle .\ dx\ d_1 + dy\ d_2 \rangle \end{aligned} \tag{5.2}$$

Some types have been annotated in grey; this is mostly for readability: in general, these types can be inferred from context. As a convention, variables on the right-hand side of equations like these that do not occur on the left-hand side are assumed fresh.

[5] We use some syntactic sugar: **let** $\langle x, y \rangle = s$ **in** t is short for **let** $z = s$ **in let** $x =$ fst z **in let** $y =$ snd z **in** t, where z is fresh. Furthermore, **let** $a = s_1; b = s_2$ **in** t is short for **let** $a = s_1$ **in let** $b = s_2$ **in** t.

The instantiated meta-type of the transformation for products is as follows:

$$\Gamma \vdash t : \sigma \times \tau \quad \rightsquigarrow \quad \Gamma \vdash \mathcal{D}_\Gamma[t] : (\sigma \times \tau) \times (\mathcal{D}[\sigma \times \tau]_2 \multimap \mathcal{D}[\Gamma]_2)$$

Because we set $\mathcal{D}[\sigma\times\tau]_2 = \mathcal{D}[\sigma]_2\times\mathcal{D}[\tau]_2$, the backpropagator in Eq. (5.2) indeed takes a pair of cotangents (d_1 and d_2), which it can pass on to the backpropagators of s and t. We have written the lambda abstraction for the backpropagator using $\underline{\lambda}$ instead of λ to emphasise that it is supposed to be a linear function.

An important observation here is the use of $(+)$ on cotangents (in this case of type $\mathcal{D}[\Gamma]_2$). All cotangent types in CHAD are commutative monoids under 0 and $(+)$; in fact, they are vector spaces, but as CHAD uses neither scalar multiplication nor negation, commutative monoids suffice. Addition on cotangents is used whenever two subterms may both contribute to the same bindings (as happens in $\mathcal{D}_\Gamma[\langle s, t\rangle]$, where both s and t potentially use the bindings in Γ); the zero is used whenever bindings are unused.[6] This happens, for example, for unit introduction (recall that $\mathcal{D}[\mathbf{1}]_2 = \mathbf{1}$):

$$\mathcal{D}_\Gamma[\langle\rangle] = \langle\langle\rangle, \underline{\lambda}\langle\rangle.\ 0_{\mathcal{D}[\Gamma]_2}\rangle \tag{5.3}$$

These occurrences of $(+)$ and 0 were foretold by the discussion on the reverse derivative of sharing and dropping on page 34.[7]

There is not a lot happening in Eq. (5.2): given the result types of $\mathcal{D}_\Gamma[s]$ and $\mathcal{D}_\Gamma[t]$ and the type that we must produce, and realising that it is probably a bad idea to leave some values unused or to duplicate them without cause, what is written in Eq. (5.2) is essentially all we can do. Similarly, Eq. (5.3) is the only reasonable thing that typechecks, given that the backpropagator must be a linear function.

Before we continue, let us also look at how variables are differentiated:

$$\begin{aligned}
&\mathcal{D}_\Gamma[x : \tau] = \langle x : \tau, \underline{\lambda} d.\ \mathbf{one}_{x\in\Gamma}\ d\rangle \\
&\mathcal{D}_\Gamma[\mathbf{let}\ x : \sigma = s\ \mathbf{in}\ t : \tau] \\
&\qquad = \mathbf{let}\ \langle x : \sigma, dx : \mathcal{D}[\sigma]_2 \multimap \mathcal{D}[\Gamma]_2\rangle = \mathcal{D}_\Gamma[s] \\
&\qquad\qquad \langle y : \tau, dy : \mathcal{D}[\tau]_2 \multimap \mathcal{D}[\Gamma, x : \sigma]_2\rangle = \mathcal{D}_{\Gamma,x:\sigma}[t] \\
&\qquad\quad \mathbf{in}\ \langle y, \underline{\lambda} d.\ \mathbf{let}\ \langle d_\Gamma, d_x\rangle = dy\ d\ \mathbf{in}\ d_\Gamma + dx\ d_x\rangle
\end{aligned} \tag{5.4}$$

where **one** constructs a *one-hot vector*, a big tuple of zeros with one non-zero value at the specified position:

$$\mathbf{one}_{x\in(\Gamma,x:\tau)}\ d = \langle 0_{\mathcal{D}[\Gamma]_2}, d\rangle \qquad \mathbf{one}_{x\in(\Gamma,y:\sigma)}\ d = \langle \mathbf{one}_{x\in\Gamma}\ d, 0_{\mathcal{D}[\sigma]_2}\rangle$$

[6] One can say that reverse differentiation sends the deletion–copying comonoid implicit in the syntax of the lambda calculus to a zero–plus monoid of cotangents.

[7] For concerns about efficiency of 0 and (+), see Chapter 6.

Because we always use $\mathbf{one}_{x \in \Gamma}$ when x is indeed present in Γ, there is no case for $\mathbf{one}_{x \in \varepsilon}$. The appearance of a one-hot vector for variable references corresponds to the fact that a term "x" uses precisely one binding from the environment (to which its cotangent is backpropagated); the other bindings are unused and hence receive zero as their cotangent contribution.

For the rule for let-bindings, note that the result of $dy\ d$ is indeed a pair because we defined $\mathcal{D}[\![\Gamma, x : \sigma]\!]_2 = \mathcal{D}[\![\Gamma]\!]_2 \times \mathcal{D}[\![\sigma]\!]_2$. The contribution from the derivative of t to the binding x goes through the environment contribution of t, because x lives in t's environment.

In the rule for let-bindings we also clearly see the expected reversal of program execution in the reverse derivative: where the source program evaluated s first and then t (at least when using call-by-value), the linear backpropagator function first calls dy (t's backpropagator) and only afterwards calls dx (s's backpropagator).

Now that we have seen some of the basic structure of the transformation, let us proceed with the next part of the source language: coproducts.

5.1.3 Dependently-typed cotangents

Coproducts make life somewhat more complicated. Indeed, suppose that we have a function $f : \sigma \to \tau_1 \sqcup \tau_2$ that, when evaluated at some input $x : \sigma$, produces the result 'inl y'. If we set $\mathcal{D}[\![\sigma \sqcup \tau]\!]_2 = \mathcal{D}[\![\sigma]\!]_2 \sqcup \mathcal{D}[\![\tau]\!]_2$, then the reverse derivative of f ought to accept any value of type $\mathcal{D}[\![\tau_1]\!]_2 \sqcup \mathcal{D}[\![\tau_2]\!]_2$, including 'inr d' for some $d : \mathcal{D}[\![\tau_2]\!]_2$. By wisdom 2 ("make illegal states unrepresentable"), however, this is a bad idea: if the reverse derivative of f receives 'inr d' for this x, it could do little else than return a zero gradient (which, if not entirely *wrong*, is not particularly correct either) or throw an error.[8]

In a language with variably-sized arrays, one gets a similar problem: the cotangent to an array of length 100 had better have length 100 itself. In fact, this is just an instance of the problem with coproducts; in a hypothetical language that allows infinitely large types, one can see a variably-sized one-dimensional array as an infinite coproduct of each possible array size:[9]

$$\coprod_{n \in \mathbb{N}} \tau^n$$

Via this analogy, passing a wrongly-sized cotangent to an array-typed term is very similar to passing a cotangent for the wrong coproduct alternative to the $f : \sigma \to \tau_1 \sqcup \tau_2$ from before.

[8] There is another problem with $\mathcal{D}[\![\sigma \sqcup \tau]\!]_2 = \mathcal{D}[\![\sigma]\!]_2 \sqcup \mathcal{D}[\![\tau]\!]_2$: it is not a monoid as there is no well-defined zero. In Chapter 6 we choose $\mathcal{D}[\![\sigma \sqcup \tau]\!]_2 = \mathbf{1} \sqcup (\mathcal{D}[\![\sigma]\!]_2 \sqcup \mathcal{D}[\![\tau]\!]_2)$ to avoid dependent types and still mostly address the monoid problem — $(+)$ is still partial, but at least there is a 0.

[9] As we are in computer science, we have $0 \in \mathbb{N}$.

CHAD maintains adherence to wisdom 2 by introducing *dependent types*: exactly what shape a cotangent value should have is dependent on the primal value it corresponds to. The cotangent to a value 'inl x' of type $\tau_1 \sqcup \tau_2$ should have type $\mathcal{D}[\![\tau_1]\!]_2$, not $\mathcal{D}[\![\tau_2]\!]_2$; the cotangent to an array-typed term should have the same length as the result of the original term. Using dependent types, we can formalise this idea as follows:

$$\begin{aligned}
&\mathcal{D}[\![-]\!]_2 : (\tau : \mathsf{Type}) \to \tau \to \mathsf{Type} \\
&\mathcal{D}[\![\mathbb{R}]\!]_2(x) = \mathbb{R} \\
&\mathcal{D}[\![\mathbb{Z}]\!]_2(n) = \mathbf{1} \\
&\mathcal{D}[\![\mathbf{1}]\!]_2(\langle\rangle) = \mathbf{1} \\
&\mathcal{D}[\![\sigma \times \tau]\!]_2(\langle x, y\rangle) = \mathcal{D}[\![\sigma]\!]_2(x) \times \mathcal{D}[\![\tau]\!]_2(y) \\
&\mathcal{D}[\![\sigma \sqcup \tau]\!]_2(\mathrm{inl}\ x) = \mathcal{D}[\![\sigma]\!]_2(x) \\
&\mathcal{D}[\![\sigma \sqcup \tau]\!]_2(\mathrm{inr}\ y) = \mathcal{D}[\![\tau]\!]_2(y)
\end{aligned} \tag{5.5}$$

In the kind signature of $\mathcal{D}[\![-]\!]_2$ we use Agda[10] syntax as sugar for Π-types; the equivalent type theory notation is $\mathcal{D}[\![-]\!]_2 : \Pi_{\tau:\mathsf{Type}}\, \Pi_{x:\tau}\, \mathsf{Type}$. A note to Haskell programmers: in Agda syntax, $f : (a : b) \to c$ does not mean that the argument of f has type a, but instead that the argument of f has type b, and that this argument's *value* is in scope with the name 'a' inside c. This allows us to refer to τ, the first argument of $\mathcal{D}[\![-]\!]_2$, inside '$\tau \to \mathsf{Type}$' in the signature of $\mathcal{D}[\![-]\!]_2$. In other words, the kind of $\mathcal{D}[\![\tau]\!]_2$ *depends* on τ; this is what "dependent" in "dependent type" refers to.

Note that the cases for reals, integers, units and products are unchanged from Section 5.1.1 apart from passing along the primal value; this means that the simply-typed rules from Section 5.1.2 for the related terms stay essentially unchanged as well.

Using similar abuse of notation as before, we extend this $\mathcal{D}[\![-]\!]_2$ to operate on environments:

$$\mathcal{D}[\![\varepsilon]\!]_2 = \mathbf{1} \qquad \mathcal{D}[\![\Gamma, x : \tau]\!]_2 = \mathcal{D}[\![\Gamma]\!]_2 \times \mathcal{D}[\![\tau]\!]_2(x) \tag{5.6}$$

The meta-type of the transformation changes as follows:

$$\Gamma \vdash t : \tau \quad \rightsquigarrow \quad \Gamma \vdash \mathcal{D}_\Gamma[t] : \tau \times (\mathcal{D}[\![\tau]\!]_2(\mathrm{fst}\ \mathcal{D}_\Gamma[t]) \multimap \mathcal{D}[\![\Gamma]\!]_2) \tag{5.7}$$

making explicit the dependency of the cotangent types on the computed values. In this dependent type judgement, the bindings in the environment Γ scope not only over the term but also over the type on the right-hand side: "fst $\mathcal{D}_\Gamma[t]$" and "$\mathcal{D}[\![\Gamma]\!]_2$" both refer to bindings in Γ.

A subtle point here is that the cotangent accepted by the differentiated term refers to the *primal*, fst $\mathcal{D}_\Gamma[t]$, rather than the original term t. They have the

[10] [Norell 2007]; `https://wiki.portal.chalmers.se/agda/pmwiki.php`

same type, and indeed the same value and hence the same coproduct branch (if applicable), so it may seem that it does not matter which we pick. However, we must take the primal in order for the typing to work out in $\mathcal{D}_\Gamma[\mathbf{case}]$; see Section 5.1.5 for details.

To see the dependence in action, let us look at coproduct introduction:

$$\begin{aligned}\mathcal{D}_\Gamma[\text{inl } s : \sigma \sqcup \tau] = \mathbf{let}\ &\langle x : \sigma, dx : \mathcal{D}[\sigma]_2(x) \multimap \mathcal{D}[\Gamma]_2\rangle = \mathcal{D}_\Gamma[s] \\ \mathbf{in}\ &\langle \text{inl } x, \underline{\lambda}(d : \mathcal{D}[\sigma \sqcup \tau]_2(\text{inl } x)).\ dx\ d\rangle\end{aligned}$$

We study 'inl' only, because $\mathcal{D}_\Gamma[\text{inr } t]$ is naturally completely analogous. In particular, let us look closely at the typing of dx and d, as they show how the dependent typing makes this rule, and in particular the application '$dx\ d$' in the backpropagator, typecheck. Since we have $\Gamma \vdash s : \sigma$, we get:

$$\Gamma \vdash \mathcal{D}_\Gamma[s] : \sigma \times (\mathcal{D}[\sigma]_2(\text{fst } \mathcal{D}_\Gamma[s]) \multimap \mathcal{D}[\Gamma]_2)$$

and hence the argument of dx has type $\mathcal{D}[\sigma]_2(\text{fst } \mathcal{D}_\Gamma[s])$. In the body of the let-binding, where x is in scope and indeed bound to 'fst $\mathcal{D}_\Gamma[s]$', we can simplify the type of the argument of dx to $\mathcal{D}[\sigma]_2(x)$. For conciseness, we have applied this simplification already in the type annotation in the left-hand side of the let-binding, even if technically x is only in scope *after* that binding.

Now, the argument (d) of the backpropagator for 'inl s' has type $\mathcal{D}[\sigma \sqcup \tau]_2(\text{fst } \mathcal{D}_\Gamma[\text{inl } s])$ by directly applying Eq. (5.7). As we have that fst $\mathcal{D}_\Gamma[\text{inl } s] = \text{inl } x$ in the body of the let-binding, we can simplify this type to $\mathcal{D}[\sigma \sqcup \tau]_2(\text{inl } x)$ as written in the type annotation. This in turn simplifies to $\mathcal{D}[\sigma]_2(x)$ by the definition of $\mathcal{D}[-]_2$ in Eq. (5.5), and this now matches with the argument type of dx, showing that the application '$dx\ d$' is indeed type-correct.

On a higher level, what we have achieved with the dependent typing is that $\mathcal{D}_\Gamma[\text{inl } s]$ knows, without having to check anything at runtime, that it receives a cotangent appropriate for passing on to the backpropagator of s. Had we set $\mathcal{D}[\sigma \sqcup \tau]_2 = \mathcal{D}[\sigma]_2 \sqcup \mathcal{D}[\tau]_2$ instead, this would not have been the case.

5.1.4 Functions: non-trivial primal types

The last complication to the typing of CHAD comes from the support for first-class functions. To see what the problem is, let us try to write the rule for function application without yet having defined $\mathcal{D}[\sigma \to \tau]_2$. As usual, we start by differentiating the subterms, and return a primal–backpropagator pair:

$$\begin{aligned}&\mathcal{D}_\Gamma[(s : \sigma \to \tau)\ (t : \sigma)] = \\ &\quad \mathbf{let}\ \langle f : \sigma \to \tau, df : \mathcal{D}[\sigma \to \tau]_2(f) \multimap \mathcal{D}[\Gamma]_2\rangle = \mathcal{D}_\Gamma[s] \\ &\qquad \langle x : \sigma, dx : \mathcal{D}[\sigma]_2(x) \multimap \mathcal{D}[\Gamma]_2\rangle = \mathcal{D}_\Gamma[t] \\ &\quad \mathbf{in}\ \langle f\ x, \underline{\lambda}(d : \mathcal{D}[\tau]_2(f\ x)).\ df\ ?\ +\ dx\ ?\rangle\end{aligned}$$

While it is unsurprising that we do not know what to pass to df, with $\mathcal{D}[\sigma \to \tau]_2$ yet undefined, a more immediate concern is what to pass to dx. In the reverse pass of a function application, we need to be able to reverse-differentiate through the function being applied, taking a cotangent for the result of the application to a cotangent for the argument of the application. But we have nothing here to accomplish that: aside from the original function f, all we have is the backpropagator df, but regardless of what we define $\mathcal{D}[\sigma \to \tau]_2$ as, df is unlikely to give us the $\mathcal{D}[\sigma]_2(x)$ we need.

It is also not possible to differentiate through the body of f here in the rule for application, because we may not have access to such a body at all: the term s could be a free variable reference! Indeed, if we are to differentiate this term:

$$f : \mathbb{R} \to \mathbb{R} \vdash f\ 42 : \mathbb{R}$$

then we are completely stuck. The only way to proceed is if the context was somehow required to not only provide the original function f, but also its derivative.

The solution of CHAD is to do precisely that: make not only cotangent types but also *primal* types different from source program types, by augmenting them with more information so that we do have what we need. Thus, alongside $\mathcal{D}[-]_2$, we now also define $\mathcal{D}[-]_1$ on types and on environments, so that we can write: (changes compared to Eq. (5.7) highlighted)

$$\Gamma \vdash t : \tau \quad \rightsquigarrow \quad \mathcal{D}[\Gamma]_1 \vdash \mathcal{D}_\Gamma[t] : \mathcal{D}[\tau]_1 \times (\mathcal{D}[\tau]_2(\mathrm{fst}\ \mathcal{D}_\Gamma[t]) \multimap \mathcal{D}[\Gamma]_2)$$

This is the final, complete form of the CHAD meta-type.

To make this work, we have to re-type $\mathcal{D}[-]_2$: (change compared to Eq. (5.5) highlighted)

$$\mathcal{D}[-]_2 : (\tau : \mathsf{Type}) \to \mathcal{D}[\tau]_1 \to \mathsf{Type}$$

because the primal argument we pass it (in '$\mathcal{D}[\tau]_2(\mathrm{fst}\ \mathcal{D}_\Gamma[t])$') is now of type $\mathcal{D}[\tau]_1$, not simply of type τ.

The equations for $\mathcal{D}[-]_1$ are as follows (trivial cases set in grey):[11]

$$\mathcal{D}[\mathbb{R}]_1 = \mathbb{R} \qquad \mathcal{D}[\mathbb{Z}]_1 = \mathbb{Z} \qquad \mathcal{D}[\mathbf{1}]_1 = \mathbf{1} \qquad \mathcal{D}[\sigma \times \tau]_1 = \mathcal{D}[\sigma]_1 \times \mathcal{D}[\tau]_1$$
$$\mathcal{D}[\sigma \sqcup \tau]_1 = \mathcal{D}[\sigma]_1 \sqcup \mathcal{D}[\tau]_1$$
$$\mathcal{D}[\sigma \to \tau]_1 = (x : \mathcal{D}[\sigma]_1) \to \Sigma_{y:\mathcal{D}[\tau]_1}\ (\mathcal{D}[\tau]_2(y) \multimap \mathcal{D}[\sigma]_2(x))$$

and contrary to $\mathcal{D}[-]_2$, $\mathcal{D}[-]_1$ maps elementwise over typing environments:

$$\mathcal{D}[\varepsilon]_1 = \varepsilon \qquad \mathcal{D}[\Gamma, x : \tau]_1 = \mathcal{D}[\Gamma]_1, x : \mathcal{D}[\tau]_1$$

[11] Without dependencies we would have $\mathcal{D}[\sigma \to \tau]_1 = \mathcal{D}[\sigma]_1 \to (\mathcal{D}[\tau]_1 \times (\mathcal{D}[\tau]_2 \multimap \mathcal{D}[\sigma]_2))$.

When restricted to types that do not contain functions, $\mathcal{D}[-]_1$ is the identity, so yet again, the rules we have so far, including the cases for $\mathcal{D}[-]_2$ in Eq. (5.5), do not materially change — only their typing changes to mention $\mathcal{D}[-]_1$ in the appropriate places.

On function types, however, $\mathcal{D}[-]_1$ does act: $\mathcal{D}[\sigma \to \tau]_1$ records not only the original function, but also the reverse derivative with respect to its argument. The 'Σ' notation is a *sigma type* or *dependent pair* (sometimes also called *dependent sum*): values of type $\Sigma_{x:\sigma}\ \tau$ are pairs of a σ value and a τ value; the difference with product types ($\times$) is that the *type* of the second component of the pair (τ) may refer to the *value* in the first component (x). As such, $\Sigma_{\_:\sigma}\ \tau$ is isomorphic to $\sigma \times \tau$. Alternatively, one can see $\Sigma_{x:\sigma}\ \tau$ as a large sum type, with not just 2 options but with one option for each value of type σ; the example supporting this perspective is $\Sigma_{x:\mathsf{Bool}}$ (**if** x **then** σ **else** τ), which is isomorphic to $\sigma \sqcup \tau$.

Differentiating functions. We need a final insight to discover what $\mathcal{D}[\sigma \to \tau]_2$ should be. Here are the term transformation rules for lambda abstraction and application, as far as we can write them at this point:

$$
\begin{array}{l}
\mathcal{D}_\Gamma[\lambda(x : \sigma).\ t : \tau] = \\
\quad \textbf{let}\ f = \lambda(x : \mathcal{D}[\sigma]_1).\ \mathcal{D}_{\Gamma,x:\sigma}[t] : \mathcal{D}[\tau]_1 \times (\mathcal{D}[\tau]_2(\mathrm{fst}\ \mathcal{D}_{\Gamma,x:\sigma}[t]) \\
\qquad\qquad\qquad\qquad\qquad\qquad \multimap \mathcal{D}[\Gamma, x : \sigma]_2) \\
\qquad f_p = \lambda(x : \mathcal{D}[\sigma]_1). \\
\qquad\qquad \textbf{let}\ \langle y : \mathcal{D}[\tau]_1, dy : \mathcal{D}[\tau]_2(y) \multimap \mathcal{D}[\Gamma]_2 \times \mathcal{D}[\sigma]_2(x)\rangle = f\ x \qquad ① \\
\qquad\qquad \textbf{in}\ \langle y, \underline{\lambda}(d : \mathcal{D}[\tau]_2(y)).\ \mathrm{snd}\ (dy\ d) : \mathcal{D}[\sigma]_2(x)\rangle \qquad ② \\
\quad \textbf{in}\ \langle f_p, \underline{\lambda}(d : \mathcal{D}[\sigma \to \tau]_2(f_p)).\ \underline{\mathrm{fst}\ (\mathrm{snd}\ (f\ ?)\ ?)} : \mathcal{D}[\Gamma]_2\rangle \qquad ③ \\
\mathcal{D}_\Gamma[(s : \sigma \to \tau)\ (t : \sigma)] = \\
\quad \textbf{let}\ \langle f : \mathcal{D}[\sigma \to \tau]_1, df : \mathcal{D}[\sigma \to \tau]_2(f) \multimap \mathcal{D}[\Gamma]_2\rangle = \mathcal{D}_\Gamma[s] \\
\qquad \langle x : \mathcal{D}[\sigma]_1, dx : \mathcal{D}[\sigma]_2(x) \multimap \mathcal{D}[\Gamma]_2\rangle = \mathcal{D}_\Gamma[t] \\
\qquad \langle y : \mathcal{D}[\tau]_1, dy : \mathcal{D}[\tau]_2(y) \multimap \mathcal{D}[\sigma]_2(x)\rangle = f\ x \\
\quad \textbf{in}\ \langle y, \underline{\lambda}(d : \mathcal{D}[\tau]_2(y)).\ df\ ? + dx\ (dy\ d)\rangle \qquad ④
\end{array}
$$

Writing out many of the types makes the rules verbose but hopefully easier to follow. The underwaved expression typechecks so far, but will need revision later.

In $\mathcal{D}_\Gamma[\lambda x.\ t]$, we need to return a pair of a primal of type $\mathcal{D}[\sigma \to \tau]_1$ and a backpropagator of type $\mathcal{D}[\sigma \to \tau]_2 \multimap \mathcal{D}[\Gamma]_2$. The primal is itself a function (f_p); it takes an $x : \mathcal{D}[\sigma]_1$ and starts by passing it to f at ① to obtain a primal output (y) and a backpropagator (dy) through the lambda body to all of its inputs: its direct argument as well as its context (free variables). On line ②, the output y is returned as the first component of f_p's result, and the argument backpropagator simply applies dy to the cotangent for the result and uses 'snd' to throw away the contributions to the context, leaving just the argument cotangent (of type $\mathcal{D}[\sigma]_2(x)$).

In the backpropagator for a lambda abstraction (the $\underline{\lambda}$ at ③), we receive a yet undefined $\mathcal{D}[\sigma \to \tau]_2$ and are supposed to produce the contributions to the environment as a result of this lambda abstraction. What are those contributions? They are precisely the $\mathcal{D}[\Gamma]_2$, produced by f at ① at each application site of the lambda, that we recklessly discarded with a 'snd' at ②!

However, retaining the environment cotangent ($\mathcal{D}[\Gamma]_2$) produced by the application $dy\ d$ at ② by returning it somehow from $f_p : \mathcal{D}[\sigma \to \tau]_1$ is difficult, since Γ here is the environment of the *lambda term*, not of the usage site of the function, and different functions of the same type $\sigma \to \tau$ may have unrelated definition-site environments. Hence, we cannot simply redefine $\mathcal{D}[\sigma \to \tau]_1$ to also return this environment cotangent (so that it could e.g. be passed to df at ④): the requisite 'Γ' is not in scope, and could not be.

Since we cannot compute the $\mathcal{D}[\Gamma]_2$ in the lambda primal (①) and communicate it through the rule for application back to the lambda backpropagator (④) to be finally used at ③, CHAD chooses to *recompute* it at ③.[12] To do this, we need to be able to differentiate through the lambda body *again* for each application of the lambda that has occurred, using the respective $\mathcal{D}[\sigma]_1$ and $\mathcal{D}[\tau]_2$ of each of those applications, when backpropagation finally reaches the lambda term itself. Therefore, in $\mathcal{D}_\Gamma[s\ t]$, we need to store the primal–cotangent pair of an application somehow so that at ③, we receive a list of pairs corresponding to all applications that have occurred.

Where do we store those (dependent) pairs? In $\mathcal{D}[\sigma \to \tau]_2$, which we now define:

$$\mathcal{D}[\sigma \to \tau]_2(f) = \text{List}\left(\Sigma_{x:\mathcal{D}[\sigma]_1}\ \mathcal{D}[\tau]_2(\text{fst}\ (f\ x))\right)$$

Note that since the primal argument f to $\mathcal{D}[\sigma \to \tau]_2$ has type $\mathcal{D}[\sigma \to \tau]_1$, we need to use 'fst' to project out the appropriate $\mathcal{D}[\tau]_1$.

We have defined $\mathcal{D}[\sigma \to \tau]_2(f)$ as a list of dependent pairs here, but more correctly it ought to be a *(dependent) copower*,[13] which imposes some additional semantic equalities on the list and imbues it with a special monoid structure. Intuitively, it should really be seen as an (infinite) map from all possible argument primals to the sum of the corresponding output cotangents — a primal maps to the empty sum, i.e. zero, if the function was never invoked with that argument. Our list is merely a practical, sparse implementation of that infinite map that

[12] This recomputation reuses a value and hence goes against the principle of "reasonable" CHAD rules that should not drop or duplicate without good reason. Perhaps fittingly, it results in a rather significant complexity issue, which we detail and fix in Section 6.8. The solution there can be seen as working around the mentioned out-of-scopeness of 'Γ' by simply putting it under an existential and giving an external proof that the loss of typing does not result in runtime errors.

[13] See Example 5.3 and Section 9 of [Vákár and Smeding 2022]. While this solution to the environment cotangent problem may feel ad hoc, mathematically it follows from the exponential structure on the appropriate category; for the interested reader: see [Vákár and Smeding 2022, §6].

neglects to store most zero-cotangent pairs. We can summarise the requirements in an equivalence relation $\simeq$ on lists, generated by the following equations:

$$\begin{aligned}
\ell_1 +\!\!+ \ell_2 +\!\!+ \ell_3 &\simeq \ell_1 +\!\!+ \ell_3 +\!\!+ \ell_2 && \text{(ordering does not matter)}\\
\ell +\!\!+ [(x, 0)] &\simeq \ell && \text{(zero-cotangent pairs do nothing)}\\
\ell +\!\!+ [(x, d_1), (x, d_2)] &\simeq \ell +\!\!+ [(x, d_1 + d_2)] && \text{(equal-primal pairs may be added)}
\end{aligned}$$

The list in $\mathcal{D}[\sigma \to \tau]_2(f)$ should be interpreted modulo $\simeq$, and its monoid structure is then simply $([], +\!\!+)$, which can be checked to respect $\simeq$.

The eliminator for copowers is a fold: (for monoids $B(x)$ and C)

$$\begin{aligned}
&\text{copowfold} : ((x : A) \to B(x) \multimap C) \to \text{List}\left(\Sigma_{x:A}\, B(x)\right) \multimap C\\
&\text{copowfold}\ f\ [] = 0\\
&\text{copowfold}\ f\ ((x, d) :: \ell) = f\ x\ d + \text{copowfold}\ f\ \ell
\end{aligned}$$

Note that if $f\ x$ is a monoid homomorphism for all x and C is a commutative monoid, then copowfold respects the equivalence relation $\simeq$ and is itself again a monoid homomorphism.

Now we can complete the definitions of $\mathcal{D}_\Gamma[\lambda x.\ t]$ and $\mathcal{D}_\Gamma[s\ t]$:

$$\begin{aligned}
&\mathcal{D}_\Gamma[\lambda(x : \sigma).\ t : \tau] =\\
&\quad \cdots\\
&\quad , \underline{\lambda}(d : \mathcal{D}[\sigma \to \tau]_2(f_p)).\ \text{copowfold}\ (\lambda x.\ \underline{\lambda} d'.\ \text{fst}\ (\text{snd}\ (f\ x)\ d'))\ d\rangle \qquad ③\\
&\mathcal{D}_\Gamma[(s : \sigma \to \tau)\ (t : \tau)] =\\
&\quad \cdots\\
&\quad \textbf{in}\ \langle y, \underline{\lambda}(d : \mathcal{D}[\tau]_2(y)).\ df\ [(x, d)] + dx\ (dy\ d)\rangle \qquad ④
\end{aligned}$$

Recall that the df at ④ is a linear function produced by the large $\underline{\lambda}$ on line ③.

In function application (④), we "log" a single application in a copower; as all these singleton copowers are cotangent contributions to the same value, the normal operation of CHAD will automatically add them together, resulting in a single log containing all applications. (The next subsection contains an example showing this behaviour.) At the lambda term, we backpropagate through the body again for each stored application, this time discarding (using 'fst') the cotangent with respect to the argument and keeping the environment cotangent. The fold adds all the resulting environment cotangents together, resulting in a single result of type $\mathcal{D}[\Gamma]_2$ that is returned.

For the full definitions, see Fig. 5.4 below.

$$\mathcal{D}[-]_1 : \mathsf{Type} \to \mathsf{Type}$$
$$\mathcal{D}[\mathbb{R}]_1 = \mathbb{R} \qquad \mathcal{D}[\mathbb{Z}]_1 = \mathbb{Z} \qquad \mathcal{D}[\mathbf{1}]_1 = \mathbf{1}$$
$$\mathcal{D}[\sigma \times \tau]_1 = \mathcal{D}[\sigma]_1 \times \mathcal{D}[\tau]_1 \qquad \mathcal{D}[\sigma \sqcup \tau]_1 = \mathcal{D}[\sigma]_1 \sqcup \mathcal{D}[\tau]_1$$
$$\mathcal{D}[\sigma \to \tau]_1 = (x : \mathcal{D}[\sigma]_1) \to \Sigma_{y:\mathcal{D}[\tau]_1} \left(\mathcal{D}[\tau]_2(y) \multimap \mathcal{D}[\sigma]_2(x)\right)$$

$$\mathcal{D}[-]_2 : (\tau : \mathsf{Type}) \to \mathcal{D}[\tau]_1 \to \mathsf{Type}$$
$$\mathcal{D}[\mathbb{R}]_2(x) = \mathbb{R}$$
$$\mathcal{D}[\mathbb{Z}]_2(n) = \mathbf{1}$$
$$\mathcal{D}[\mathbf{1}]_2(\langle\rangle) = \mathbf{1}$$
$$\mathcal{D}[\sigma \times \tau]_2(\langle x, y\rangle) = \mathcal{D}[\sigma]_2(x) \times \mathcal{D}[\tau]_2(y)$$
$$\mathcal{D}[\sigma \sqcup \tau]_2(\mathrm{inl}\ x) = \mathcal{D}[\sigma]_2(x)$$
$$\mathcal{D}[\sigma \sqcup \tau]_2(\mathrm{inr}\ y) = \mathcal{D}[\tau]_2(y)$$
$$\mathcal{D}[\sigma \to \tau]_2(f) = \mathrm{List}\ \left(\Sigma_{x:\mathcal{D}[\sigma]_1} \mathcal{D}[\tau]_2(\mathrm{fst}\ (f\ x))\right)$$

$$\mathcal{D}[\varepsilon]_1 = \varepsilon \qquad \mathcal{D}[\varepsilon]_2 = \mathbf{1}$$
$$\mathcal{D}[\Gamma, x : \tau]_1 = \mathcal{D}[\Gamma]_1, x : \mathcal{D}[\tau]_1 \qquad \mathcal{D}[\Gamma, x : \tau]_2 = \mathcal{D}[\Gamma]_2 \times \mathcal{D}[\tau]_2(x)$$

Figure 5.2: The naive CHAD transformation on types and environments.

5.1.5 Full transformation

The full transformation on the source language in Fig. 5.1 is spread over three figures (pages 218 to 221): on types in Fig. 5.2 and on terms in Figs. 5.3 and 5.4.[14] As before, some types are annotated inside the transformation; the choice for which types to show and which to leave implicit is based on clarity and readability of the layout, not on anything technical.

To get a feeling for how the rules interact, we will first look at the derivative of two example terms and how their data flow illustrates the general operation of CHAD-differentiated programs. Afterwards, in Section 5.1.6, we will briefly go over the rules in Figs. 5.3 and 5.4 individually to stress the important features.

Example: let-binding. Consider the term $a : \mathbb{R} \vdash t_1 : \mathbb{R}$ given by:

$$t_1 = \mathbf{let}\ b = \langle a, 2 \cdot a\rangle \\ \mathbf{in}\ \mathrm{fst}\ b + \mathrm{snd}\ b$$

[14]The rules in these figures have been type-checked by means of a direct translation of the rules to Agda. As (deeply) embedding dependently-typed languages is rather cumbersome, this is easiest when the target language is chosen to be Agda itself: while the "transformation" does take a term, it would not produce one but instead compute the answer directly.

We write concrete primitive operations, e.g. '$a + b$', instead of $\mathrm{op}_{\mathbb{R}}(a, b)$ to ease interpretation. Algebraically simplifying t_1 yields $t_1 = a + 2 \cdot a = 3 \cdot a$, so we expect that $\mathcal{D}_{a:\mathbb{R}}[t_1]$ algebraically simplifies to $\langle 3 \cdot a, \underline{\lambda} d.\ 3 \cdot d \rangle$.[15]

The CHAD reverse derivative, $a : \mathbb{R} \vdash \mathcal{D}_{\varepsilon,a:\mathbb{R}}[t_1] : \mathbb{R} \times (\mathbb{R} \multimap (\mathbf{1} \times \mathbb{R}))$, is:[16]

$$
\begin{array}{l}
\textbf{let } \langle b, db \rangle = \\
\quad \textbf{let } \langle x, dx \rangle = \langle a, \underline{\lambda} d.\ \mathbf{one}_{a \in (\varepsilon, a:\mathbb{R})}\ d \rangle \\
\quad\quad \langle y, dy \rangle = \textbf{let } \langle x_1, dx_1 \rangle = \langle 2, \underline{\lambda} d.\ 0 \rangle; \langle x_2, dx_2 \rangle = \langle a, \underline{\lambda} d.\ \mathbf{one}_{a \in (\varepsilon, a:\mathbb{R})}\ d \rangle \\
\quad\quad\quad\quad \textbf{in } \langle x_1 \cdot x_2, \underline{\lambda} d.\ dx_1\ (d \cdot x_2) + dx_2\ (d \cdot x_1) \rangle \\
\quad \textbf{in } \langle \langle x, y \rangle, \underline{\lambda} \langle d_1, d_2 \rangle.\ dx\ d_1 + dy\ d_2 \rangle \\
\quad \langle y, dy \rangle = \\
\quad\quad \textbf{let } \langle x_1, dx_1 \rangle = \textbf{let } \langle x, dx \rangle = \langle b, \underline{\lambda} d.\ \mathbf{one}_{b \in (\varepsilon, a:\mathbb{R}, b:\mathbb{R} \times \mathbb{R})}\ d \rangle \\
\quad\quad\quad\quad\quad\quad \textbf{in } \langle \mathrm{fst}\ x, \underline{\lambda} d.\ dx\ \langle d, 0 \rangle \rangle \\
\quad\quad\quad\quad \langle x_2, dx_2 \rangle = \textbf{let } \langle x, dx \rangle = \langle b, \underline{\lambda} d.\ \mathbf{one}_{b \in (\varepsilon, a:\mathbb{R}, b:\mathbb{R} \times \mathbb{R})}\ d \rangle \\
\quad\quad\quad\quad\quad\quad \textbf{in } \langle \mathrm{snd}\ x, \underline{\lambda} d.\ dx\ \langle 0, d \rangle \rangle \\
\quad\quad \textbf{in } \langle x_1 + x_2, \underline{\lambda} d.\ dx_1\ d + dx_2\ d \rangle \\
\textbf{in } \langle y, \underline{\lambda} d.\ \textbf{let } \langle d_\Gamma, d_x \rangle = dy\ d \textbf{ in } d_\Gamma + db\ d_x \rangle
\end{array}
$$

The derivatives of variable references are easy to recognise by their use of **one**, which, as stated before, produce a one-hot vector:

$$
\begin{array}{ll}
\mathbf{one}_{a \in (\varepsilon, a:\mathbb{R}, b:\mathbb{R} \times \mathbb{R})}\ d = \langle \langle 0_{\mathcal{D}[\varepsilon]_2}, d \rangle, 0_{\mathcal{D}[\mathbb{R} \times \mathbb{R}]_2(b)} \rangle & = \langle \langle \langle \rangle, d \rangle, 0_{\mathbb{R} \times \mathbb{R}} \rangle \\
\mathbf{one}_{b \in (\varepsilon, a:\mathbb{R}, b:\mathbb{R} \times \mathbb{R})}\ d = \langle 0_{\mathcal{D}[\varepsilon, a:\mathbb{R}]_2}, d \rangle & = \langle \langle \langle \rangle, 0_{\mathbb{R}} \rangle, d \rangle
\end{array}
$$

Simplifying the $\mathcal{D}_{\varepsilon,a:\mathbb{R}}[\langle a, 2 \cdot a \rangle]$ and $\mathcal{D}_{\varepsilon,a:\mathbb{R},b:\mathbb{R} \times \mathbb{R}}[\mathrm{fst}\ b + \mathrm{snd}\ b]$ blocks by standard beta-reduction and keeping the rest as-is, we get:

$$
\begin{array}{ll}
\textbf{let } \langle b, db \rangle = \langle \langle a, 2 \cdot a \rangle, \underline{\lambda} \langle d_1, d_2 \rangle.\ \mathbf{one}_{a \in (\varepsilon, a:\mathbb{R})}\ d_1 + \mathbf{one}_{a \in (\varepsilon, a:\mathbb{R})}\ (d_2 \cdot 2) \rangle & ① \\
\quad \langle y, dy \rangle = \langle \mathrm{fst}\ b + \mathrm{snd}\ b & \\
\quad\quad\quad\quad , \underline{\lambda} d.\ \mathbf{one}_{b \in (\varepsilon, a:\mathbb{R}, b:\mathbb{R} \times \mathbb{R})}\ \langle d, 0 \rangle + \mathbf{one}_{b \in (\varepsilon, a:\mathbb{R}, b:\mathbb{R} \times \mathbb{R})}\ \langle 0, d \rangle \rangle & ② \\
\textbf{in } \langle y, \underline{\lambda} d.\ \textbf{let } \langle d_\Gamma, d_x \rangle = dy\ d \textbf{ in } d_\Gamma + db\ d_x \rangle &
\end{array}
$$

What is left is the skeleton of $\mathcal{D}_\Gamma[\mathbf{let}]$ filled in with simplified derivatives for the right-hand side and the body.

At this point it is clearly visible that the first component of the result of $\mathcal{D}_{\varepsilon,a:\mathbb{R}}[t_1]$ is indeed equivalent to t_1. For the backpropagators (the second components of the pairs), one should keep in mind the intuition that a backpropagator corresponding to a value x is expected to take a cotangent with respect to x, and then return the contributions to the environment as a result of backpropagating

[15] Using notation from Section 2.2.2: if t_1 implements $f : \mathbb{R} \to \mathbb{R}$, then $(Df)^\top\ a\ d = 3 \cdot d$.

[16] Because of the definition of $\mathcal{D}[\Gamma]_2$ (Eq. (5.6)), we pedantically write $\varepsilon, a : \mathbb{R}$ for an environment with 1-entry.

$$\Gamma \vdash t : \tau \quad \rightsquigarrow \quad \mathcal{D}[\Gamma]_1 \vdash \mathcal{D}_\Gamma[t] : \mathcal{D}[\tau]_1 \times (\mathcal{D}[\tau]_2(\mathrm{fst}\ \mathcal{D}_\Gamma[t]) \multimap \mathcal{D}[\Gamma]_2)$$

$$\mathcal{D}_\Gamma[x : \tau] = \langle x : \mathcal{D}[\tau]_1, \underline{\lambda} d.\ \mathbf{one}_{x \in \Gamma}\ d \rangle$$

$$\begin{aligned}
&\mathcal{D}_\Gamma[\mathbf{let}\ x : \sigma = s\ \mathbf{in}\ t : \tau] \\
&\qquad = \mathbf{let}\ \langle x : \mathcal{D}[\sigma]_1, dx : \mathcal{D}[\sigma]_2(x) \multimap \mathcal{D}[\Gamma]_2 \rangle = \mathcal{D}_\Gamma[s] \\
&\qquad\qquad \langle y : \mathcal{D}[\tau]_1, dy : \mathcal{D}[\tau]_2(y) \multimap \mathcal{D}[\Gamma, x : \sigma]_2 \rangle = \mathcal{D}_{\Gamma, x:\sigma}[t] \\
&\qquad\quad \mathbf{in}\ \langle y, \underline{\lambda} d.\ \mathbf{let}\ \langle d_\Gamma, d_x \rangle = dy\ d\ \mathbf{in}\ d_\Gamma + dx\ d_x \rangle
\end{aligned}$$

$$\mathcal{D}_\Gamma[\langle\rangle] \quad = \langle \langle\rangle, \underline{\lambda} \langle\rangle.\ 0_{\mathcal{D}[\Gamma]_2} \rangle$$

$$\begin{aligned}
&\mathcal{D}_\Gamma[\langle s : \sigma, t : \tau \rangle] \\
&\qquad = \mathbf{let}\ \langle x : \mathcal{D}[\sigma]_1, dx : \mathcal{D}[\sigma]_2(x) \multimap \mathcal{D}[\Gamma]_2 \rangle = \mathcal{D}_\Gamma[s] \\
&\qquad\qquad \langle y : \mathcal{D}[\tau]_1, dy : \mathcal{D}[\tau]_2(y) \multimap \mathcal{D}[\Gamma]_2 \rangle = \mathcal{D}_\Gamma[t] \\
&\qquad\quad \mathbf{in}\ \langle \langle x, y \rangle, \underline{\lambda} \langle d_1, d_2 \rangle.\ dx\ d_1 + dy\ d_2 \rangle
\end{aligned}$$

$$\begin{aligned}
&\mathcal{D}_\Gamma[\mathrm{fst}\ (t : \sigma \times \tau)] \\
&\qquad = \mathbf{let}\ \langle x : \mathcal{D}[\sigma]_1 \times \mathcal{D}[\tau]_1, dx : \mathcal{D}[\sigma \times \tau]_2(x) \multimap \mathcal{D}[\Gamma]_2 \rangle = \mathcal{D}_\Gamma[t] \\
&\qquad\quad \mathbf{in}\ \langle \mathrm{fst}\ x, \underline{\lambda} d.\ dx\ \langle d, 0_{\mathcal{D}[\tau]_2(\mathrm{snd}\ x)} \rangle \rangle
\end{aligned}$$

$$\begin{aligned}
&\mathcal{D}_\Gamma[\mathrm{snd}\ (t : \sigma \times \tau)] \\
&\qquad = \mathbf{let}\ \langle x : \mathcal{D}[\sigma]_1 \times \mathcal{D}[\tau]_1, dx : \mathcal{D}[\sigma \times \tau]_2(x) \multimap \mathcal{D}[\Gamma]_2 \rangle = \mathcal{D}_\Gamma[t] \\
&\qquad\quad \mathbf{in}\ \langle \mathrm{snd}\ x, \underline{\lambda} d.\ dx\ \langle 0_{\mathcal{D}[\sigma]_2(\mathrm{fst}\ x)}, d \rangle \rangle
\end{aligned}$$

$$\begin{aligned}
&\mathcal{D}_\Gamma[\mathrm{inl}\ t : \sigma \sqcup \tau] \\
&\qquad = \mathbf{let}\ \langle x : \mathcal{D}[\sigma]_1, dx : \mathcal{D}[\sigma]_2(x) \multimap \mathcal{D}[\Gamma]_2 \rangle = \mathcal{D}_\Gamma[t] \\
&\qquad\quad \mathbf{in}\ \langle \mathrm{inl}\ x, \underline{\lambda} (d : \mathcal{D}[\sigma \sqcup \tau]_2(\mathrm{inl}\ x)).\ dx\ d \rangle
\end{aligned}$$

$$\begin{aligned}
&\mathcal{D}_\Gamma[\mathrm{inr}\ t : \sigma \sqcup \tau] \\
&\qquad = \mathbf{let}\ \langle x : \mathcal{D}[\tau]_1, dx : \mathcal{D}[\tau]_2(x) \multimap \mathcal{D}[\Gamma]_2 \rangle = \mathcal{D}_\Gamma[t] \\
&\qquad\quad \mathbf{in}\ \langle \mathrm{inr}\ x, \underline{\lambda} (d : \mathcal{D}[\sigma \sqcup \tau]_2(\mathrm{inr}\ x)).\ dx\ d \rangle
\end{aligned}$$

$$\begin{aligned}
&\mathcal{D}_\Gamma[\mathbf{case}\ s : \sigma \sqcup \tau\ \mathbf{of}\ \{\ \mathrm{inl}\ x \to t_1 \mid \mathrm{inr}\ y \to t_2\ \} : \rho] \\
&\qquad = \mathbf{let}\ \langle u : \mathcal{D}[\sigma \sqcup \tau]_1, du : \mathcal{D}[\sigma \sqcup \tau]_2(u) \multimap \mathcal{D}[\Gamma]_2 \rangle = \mathcal{D}_\Gamma[s] \\
&\qquad\quad \mathbf{in\ case}\ u\ \mathbf{of} \\
&\qquad\qquad \mathrm{inl}\ x \to \mathbf{let}\ \langle v, dv : \mathcal{D}[\rho]_2(v) \multimap \mathcal{D}[\Gamma, x : \sigma]_2 \rangle = \mathcal{D}_{\Gamma, x:\sigma}[t_1] \\
&\qquad\qquad\qquad \mathbf{in}\ \langle v, \underline{\lambda} d.\ \mathbf{let}\ \langle d_\Gamma, d_x : \mathcal{D}[\sigma]_2(x) \rangle = dv\ d \\
&\qquad\qquad\qquad\qquad \mathbf{in}\ d_\Gamma + du\ d_x \rangle \\
&\qquad\qquad \mathrm{inr}\ y \to \mathbf{let}\ \langle v, dv : \mathcal{D}[\rho]_2(v) \multimap \mathcal{D}[\Gamma, y : \tau]_2 \rangle = \mathcal{D}_{\Gamma, y:\tau}[t_2] \\
&\qquad\qquad\qquad \mathbf{in}\ \langle v, \underline{\lambda} d.\ \mathbf{let}\ \langle d_\Gamma, d_y : \mathcal{D}[\tau]_2(y) \rangle = dv\ d \\
&\qquad\qquad\qquad\qquad \mathbf{in}\ d_\Gamma + du\ d_y \rangle
\end{aligned}$$

$$\mathbf{one}_{x \in (\Gamma, x:\tau)}\ d = \langle 0_{\mathcal{D}[\Gamma]_2}, d \rangle \qquad \mathbf{one}_{x \in (\Gamma, y:\sigma)}\ d = \langle \mathbf{one}_{x \in \Gamma}\ d, 0_{\mathcal{D}[\sigma]_2} \rangle$$

Figure 5.3: The naive CHAD transformation on terms (1).

$$
\begin{aligned}
&\mathcal{D}_\Gamma[\lambda(x:\sigma).\ t:\tau] \\
&\qquad = \mathbf{let}\ f = \lambda(x:\mathcal{D}[\sigma]_1).\ \mathcal{D}_{\Gamma,x:\sigma}[t] : \mathcal{D}[\tau]_1 \times (\mathcal{D}[\tau]_2(\mathrm{fst}\ \mathcal{D}_{\Gamma,x:\sigma}[t]) \\
&\qquad\qquad\qquad\qquad \multimap \mathcal{D}[\Gamma, x:\sigma]_2) \\
&\qquad\qquad f_p = \lambda x.\ \mathbf{let}\ \langle y : \mathcal{D}[\tau]_1, dy : \mathcal{D}[\tau]_2(y) \multimap \mathcal{D}[\Gamma, x:\sigma]_2\rangle = f\ x \\
&\qquad\qquad\qquad\qquad \mathbf{in}\ \langle y, \underline{\lambda}(d : \mathcal{D}[\tau]_2(y)).\ \mathrm{snd}\ (dy\ d) : \mathcal{D}[\sigma]_2(x)\rangle \\
&\qquad\quad \mathbf{in}\ \langle f_p, \underline{\lambda}(d : \mathcal{D}[\sigma \to \tau]_2(f_p)). \\
&\qquad\qquad\qquad\qquad \mathrm{copowfold}\ (\lambda x.\ \underline{\lambda} d'.\ \mathrm{fst}\ (\mathrm{snd}\ (f\ x)\ d'))\ d\rangle \\
&\mathcal{D}_\Gamma[(s:\sigma \to \tau)\ (t:\sigma)] \\
&\qquad = \mathbf{let}\ \langle f : \mathcal{D}[\sigma \to \tau]_1, df : \mathcal{D}[\sigma \to \tau]_2(f) \multimap \mathcal{D}[\Gamma]_2\rangle = \mathcal{D}_\Gamma[s] \\
&\qquad\qquad \langle x : \mathcal{D}[\sigma]_1, dx : \mathcal{D}[\sigma]_2(x) \multimap \mathcal{D}[\Gamma]_2\rangle = \mathcal{D}_\Gamma[t] \\
&\qquad\qquad \langle y : \mathcal{D}[\tau]_1, dy : \mathcal{D}[\tau]_2(y) \multimap \mathcal{D}[\sigma]_2(x)\rangle = f\ x \\
&\qquad\quad \mathbf{in}\ \langle y, \underline{\lambda}(d : \mathcal{D}[\tau]_2(y)).\ df\ [(x,d)] + dx\ (dy\ d)\rangle \\
&\mathcal{D}_\Gamma[r:\mathbb{R}] = \langle r, \underline{\lambda} d.\ 0_{\mathcal{D}[\Gamma]_2}\rangle \\
&\mathcal{D}_\Gamma[\mathrm{sign}\ t] \\
&\qquad = \mathbf{let}\ \langle x : \mathbb{R}, dx : \mathbb{R} \multimap \mathcal{D}[\Gamma]_2\rangle = \mathcal{D}_\Gamma[t] \\
&\qquad\quad \mathbf{in}\ \langle \mathrm{sign}\ x, \underline{\lambda} d : \mathcal{D}[\mathbf{1} \sqcup \mathbf{1}]_2(\mathrm{sign}\ x).\ 0_{\mathcal{D}[\Gamma]_2}\rangle \\
&\mathcal{D}_\Gamma[\mathrm{op}_{\mathbb{R}}(t_1, \ldots, t_n)] \\
&\qquad = \mathbf{let}\ \langle x_1, dx_1 : \mathbb{R} \multimap \mathcal{D}[\Gamma]_2\rangle = \mathcal{D}_\Gamma[t_1]; \ldots; \langle x_n, dx_n\rangle = \mathcal{D}_\Gamma[t_n] \\
&\qquad\quad \mathbf{in}\ \langle \mathrm{op}_{\mathbb{R}}(x_1, \ldots, x_n) \\
&\qquad\qquad\quad ,\underline{\lambda}(d:\mathbb{R}).\ dx_1\ (\partial_1 \mathrm{op}_{\mathbb{R}}(d; x_1, \ldots, x_n)) \\
&\qquad\qquad\qquad\qquad + \cdots + dx_n\ (\partial_n \mathrm{op}_{\mathbb{R}}(d; x_1, \ldots, x_n))\rangle \\
&\mathcal{D}_\Gamma[n] \quad = \langle n, \underline{\lambda}\langle\rangle.\ 0_{\mathcal{D}[\Gamma]_2}\rangle \\
&\mathcal{D}_\Gamma[\mathrm{op}_{\mathbb{Z}}(t_1, \ldots, t_n)] \\
&\qquad = \mathbf{let}\ \langle x_1, dx_1 : \mathbf{1} \multimap \mathcal{D}[\Gamma]_2\rangle = \mathcal{D}_\Gamma[t_1]; \ldots; \langle x_n, dx_n\rangle = \mathcal{D}_\Gamma[t_n] \\
&\qquad\quad \mathbf{in}\ \langle \mathrm{op}_{\mathbb{Z}}(x_1, \ldots, x_n), \underline{\lambda}\langle\rangle.\ 0_{\mathcal{D}[\Gamma]_2}\rangle
\end{aligned}
$$

$$
\begin{aligned}
&\mathrm{copowfold} : ((x:A) \to B(x) \multimap C) \to \mathrm{List}\ \left(\Sigma_{x:A}\ B(x)\right) \multimap C \\
&\mathrm{copowfold}\ f\ [] = 0 \\
&\mathrm{copowfold}\ f\ ((x,d) :: \ell) = f\ x\ d + \mathrm{copowfold}\ f\ \ell
\end{aligned}
$$

Figure 5.4: The naive CHAD transformation on terms (2).

through the term that produced x. In db, this does indeed happen: if the derivative of the result with respect to the term $\langle a, 2 \cdot a\rangle$ is $\langle d_1, d_2\rangle$, then the resulting cotangent with respect to a should be $d_1 + 2d_2$, and thus the environment cotangent (of type $\mathcal{D}[\Gamma]_2 = \mathcal{D}[\varepsilon, a : \mathbb{R}]_2 = \mathbf{1} \times \mathbb{R}$) should be $\langle\langle\rangle, d_1 + 2d_2\rangle$, which is indeed computed correctly at ①.

Once we are in the body of the let-binding, however, references to the right-hand side happen through a *variable reference*. Indeed, at this point, the right-hand side term ('$\langle a, 2 \cdot a\rangle$' in this case) is not any different from an input — it is just another entry in the environment. Hence, contributions to it happen through **one**, as can be seen at ②. In effect, this means that all cotangent contributions to the right-hand side of a let-binding (in this case, $\langle d, 0\rangle$ and $\langle 0, d\rangle$) are first collected and added together "at the top of the let-body", before the sum is split off the environment cotangent collector ($\mathcal{D}[\Gamma]_2$) in the backpropagator of $\mathcal{D}_\Gamma[\mathbf{let}]$ as d_x. This d_x is then sent into the backpropagator of the right-hand side (db).

Spoiler: First-order backpropagators. The design of CHAD ensures that — if one ignores function types! — every backpropagator is invoked at most once. It turns out that this insight allows us to restructure the code transformation so that it avoids creating lambda terms for the (non-function-type) backpropagators at all. To illustrate: a different way of simplifying $\mathcal{D}_{\varepsilon,a:\mathbb{R}}[t_1]$ arrives at the following, using mostly just inlining of variables without increasing code size or duplicating work:

$$\begin{aligned}
&\mathbf{let}\ b = \langle a, 2 \cdot a\rangle \\
&\mathbf{in}\ \langle \mathrm{fst}\ b + \mathrm{snd}\ b \\
&\quad ,\underline{\lambda} d.\ \mathbf{let}\ \langle d_\Gamma, d_x\rangle = \mathbf{one}_{b\in(\varepsilon,a:\mathbb{R},b:\mathbb{R}\times\mathbb{R})}\ \langle d, 0\rangle + \mathbf{one}_{b\in(\varepsilon,a:\mathbb{R},b:\mathbb{R}\times\mathbb{R})}\ \langle 0, d\rangle\rangle \\
&\qquad\quad \mathbf{in}\ d_\Gamma + \mathbf{one}_{a\in(\varepsilon,a:\mathbb{R})}\ d_x + 0 + \mathbf{one}_{a\in(\varepsilon,a:\mathbb{R})}\ (d_x \cdot 2)
\end{aligned}$$

We see the forward pass, followed by a lambda abstraction for the backpropagator, and then no more lambda abstractions inside that backpropagator. It turns out that with a change to $\mathcal{D}_\Gamma[\mathbf{case}]$, one can ensure that this simplification is *always* possible (as long as, unsurprisingly, no lambda abstraction has been used in the source program). We use this observation in Section 7.3 to guarantee, by construction, absence of lambda abstraction in the derivative of programs that did not already use lambda abstraction before.

Example: functions. Consider the term $a : \mathbb{R} \vdash t_2 : \mathbb{R}$ given by:

$$\begin{aligned}
t_2 = \mathbf{let}\ h &= \lambda v.\ a \cdot v \\
\mathbf{in}\ h\ a &+ h\ (3 \cdot a)
\end{aligned}$$

The CHAD reverse derivative, $a : \mathbb{R} \vdash \mathcal{D}_{\varepsilon,a:\mathbb{R}}[t_2] : \mathbb{R} \times (\mathbb{R} \multimap (\mathbf{1} \times \mathbb{R}))$, looks as follows, after simplifying the subexpressions corresponding to '$\mathcal{D}_{\varepsilon,a:\mathbb{R},v:\mathbb{R}}[a \cdot v]$' and '$\mathcal{D}_{\varepsilon,a:\mathbb{R},f:\mathbb{R}\to\mathbb{R}}[3 \cdot a]$' for conciseness:

$$
\begin{aligned}
&\mathbf{let}\ \langle h, dh\rangle = \mathbf{let}\ f = \lambda v.\ \langle a \cdot v, \underline{\lambda} d.\ \mathbf{one}_{a\in(\varepsilon,a:\mathbb{R},v:\mathbb{R})}\ (d \cdot v) + \mathbf{one}_{v\in(\varepsilon,a:\mathbb{R},v:\mathbb{R})}\ (d \cdot a)\rangle \\
&\qquad\qquad f_p = \lambda v.\ \mathbf{let}\ \langle y, dy\rangle = f\ v\ \mathbf{in}\ \langle y, \underline{\lambda} d.\ \mathrm{snd}\ (dy\ d)\rangle \\
&\qquad\qquad \mathbf{in}\ \langle f_p, \underline{\lambda} d.\ \mathrm{copowfold}\ (\lambda v.\ \underline{\lambda} d'.\ \mathrm{fst}\ (\mathrm{snd}\ (f\ v)\ d'))\ d\rangle \\
&\quad \langle y, dy\rangle = \mathbf{let}\ \langle x_1, dx_1\rangle = \mathbf{let}\ \langle h, dh\rangle = \langle h, \underline{\lambda} d.\ \mathbf{one}_{h\in(\varepsilon,a:\mathbb{R},h:\mathbb{R}\to\mathbb{R})}\ d\rangle \\
&\qquad\qquad\qquad\qquad\qquad\qquad \langle x, dx\rangle = \langle a, \underline{\lambda} d.\ \mathbf{one}_{a\in(\varepsilon,a:\mathbb{R},h:\mathbb{R}\to\mathbb{R})}\ d\rangle \\
&\qquad\qquad\qquad\qquad\qquad\qquad \langle y, dy\rangle = h\ x \\
&\qquad\qquad\qquad\qquad\qquad \mathbf{in}\ \langle y, \underline{\lambda} d.\ dh\ [(x, d)] + dx\ (dy\ d)\rangle \\
&\qquad\qquad \langle x_2, dx_2\rangle = \mathbf{let}\ \langle h, dh\rangle = \langle h, \underline{\lambda} d.\ \mathbf{one}_{h\in(\varepsilon,a:\mathbb{R},h:\mathbb{R}\to\mathbb{R})}\ d\rangle \\
&\qquad\qquad\qquad\qquad\qquad\qquad \langle x, dx\rangle = \langle 3 \cdot a, \underline{\lambda} d.\ \mathbf{one}_{a\in(\varepsilon,a:\mathbb{R},h:\mathbb{R}\to\mathbb{R})}\ (d \cdot 3)\rangle \\
&\qquad\qquad\qquad\qquad\qquad\qquad \langle y, dy\rangle = h\ x \\
&\qquad\qquad\qquad\qquad\qquad \mathbf{in}\ \langle y, \underline{\lambda} d.\ dh\ [(x, d)] + dx\ (dy\ d)\rangle \\
&\qquad\qquad \mathbf{in}\ \langle x_1 + x_2, \underline{\lambda} d.\ dx_1\ d + dx_2\ d\rangle \\
&\mathbf{in}\ \langle y, \underline{\lambda} d.\ \mathbf{let}\ \langle d_\Gamma, d_x\rangle = dy\ d\ \mathbf{in}\ d_\Gamma + dh\ d_x\rangle
\end{aligned}
$$

Using basic compiler optimisations, we can further simplify this to:[17]

$$
\begin{aligned}
&\mathbf{let}\ \langle h, dh\rangle = \mathbf{let}\ f = \lambda v.\ \langle a \cdot v, \underline{\lambda} d.\ \mathbf{one}_{a\in(\varepsilon,a:\mathbb{R},v:\mathbb{R})}\ (d \cdot v) + \mathbf{one}_{v\in(\varepsilon,a:\mathbb{R},v:\mathbb{R})}\ (d \cdot a)\rangle \\
&\qquad\qquad \mathbf{in}\ \langle \lambda v.\ \mathbf{let}\ \langle y, dy\rangle = f\ v\ \mathbf{in}\ \langle y, \underline{\lambda} d.\ \mathrm{snd}\ (dy\ d)\rangle \\
&\qquad\qquad\quad ,\underline{\lambda} d.\ \mathrm{copowfold}\ (\lambda v.\ \underline{\lambda} d'.\ \mathrm{fst}\ (\mathrm{snd}\ (f\ v)\ d'))\ d\rangle \\
&\quad \langle y_1, dy_1\rangle = h\ a \\
&\quad x_2 = 3 \cdot a \\
&\quad \langle y_2, dy_2\rangle = h\ x_2 \\
&\mathbf{in}\ \langle y_1 + y_2 \\
&\quad ,\underline{\lambda} d.\ \mathbf{let}\ \langle d_\Gamma, d_x\rangle = \mathbf{one}_{h\in(\varepsilon,a:\mathbb{R},h:\mathbb{R}\to\mathbb{R})}\ [(a, d)] + \mathbf{one}_{a\in(\varepsilon,a:\mathbb{R},h:\mathbb{R}\to\mathbb{R})}\ (dy_1\ d) + \\
&\qquad\qquad\qquad \mathbf{one}_{h\in(\varepsilon,a:\mathbb{R},h:\mathbb{R}\to\mathbb{R})}\ [(x_2, d)] + \mathbf{one}_{a\in(\varepsilon,a:\mathbb{R},h:\mathbb{R}\to\mathbb{R})}\ (dy_2\ d \cdot 3) \\
&\qquad \mathbf{in}\ d_\Gamma + dh\ d_x\rangle
\end{aligned}
$$

In the top-level backpropagator (the three-line linear lambda at the bottom of the simplified term), we first backpropagate through both applications of h to produce two contributions to h ($[(a, d)]$ and $[(x_2, d)] = [(3 \cdot a, d)]$) and two contributions to a ($dy_1\ d$ and $dy_2\ d \cdot 3$). The first and third addition (between the two contributions arising from a single application) come from the use of '+' in $\mathcal{D}_\Gamma[s\ t]$ in Fig. 5.4; the second addition (adding the two copowers, as well as the two contributions to a, together) comes from the use of '+' in $\mathcal{D}_\Gamma[\mathrm{op}_\mathbb{R}(t_1, \ldots, t_n)]$ in Fig. 5.4, instantiated at $n = 2$, $\mathrm{op}_\mathbb{R} = (+)$, $t_1 = h\ a$ and $t_2 = h\ (3 \cdot a)$. This is

[17] Specifically, in this case: inlining of variables used at most once, rearrangement of let-bindings, let-splitting ('$\mathbf{let}\ \langle x, y\rangle = \langle t_1, t_2\rangle\ \mathbf{in}\ t_3$' to '$\mathbf{let}\ x = t_1\ \mathbf{in}\ \mathbf{let}\ y = t_2\ \mathbf{in}\ t_3$') and beta-reduction.

nothing specific to arithmetic operations: simply because the '+' term is a term with two subterms, it adds environment cotangent contributions from those two subterms together in its own backpropagator.

In this way, the two copowers merge to $[(a, d), (3 \cdot a, d)]$; this value (d_x) then gets passed to *dh*, which folds over it with the derivative of the body of the original *h*. In this case, that yields yields two contributions to a ($d \cdot a$ and $d \cdot (3 \cdot a)$) that get added to the contributions to a in d_Γ, which arose from the passing of a as an *argument* to the applications. In all, we get four contributions to a from a run of the top-level backpropagator, which is as expected: two from the use of a in the arguments to h (in d_Γ), and two from the appearance of a in the closure of h (in $dh\ d_x$).

5.1.6 Overview of the rules

The type transformations are shown in Fig. 5.2: the primal type transformation $\mathcal{D}[-]_1$ (the identity on all but function types — see Section 5.1.4), the dual type transformation $\mathcal{D}[-]_2$ (using dependence on the primal for coproducts — see Section 5.1.3) and their liftings to environments (elementwise for $\mathcal{D}[-]_1$ and as a product for $\mathcal{D}[-]_2$).

In Figs. 5.3 and 5.4, then, we see the full term transformation. In the meta-type at the top of Fig. 5.3, note that the argument to the backpropagator refers to the primal result of the transformation, not simply t; similarly, in $\mathcal{D}[\Gamma]_2$, the uses of $\mathcal{D}[-]_2$ refer to the variables that now come from $\mathcal{D}[\Gamma]_1$, and hence are also primals as expected. The definitions of **one** and copowfold are repeated at the bottom of the figures for completeness.

Some notes about each rule individually:

- Variable references change type under $\mathcal{D}_\Gamma[-]$; their backpropagator puts the cotangent in an environment cotangent. As discussed in Section 5.1.5, uses of '+' in various other rules ensure that the environment cotangents produced by variable references in the body of a let-binding get combined before $\mathcal{D}_\Gamma[\mathbf{let}]$ receives them and passes them on.

- In $\mathcal{D}_\Gamma[\mathbf{let}]$, note that the body ($t$) is differentiated under an extended environment. The $x : \mathcal{D}[\sigma]_1$ that is required to be in scope for $\mathcal{D}_{\Gamma,x:\sigma}[t]$ is, in fact, in scope because of the binding on the line before; the additional variable dx is silently discarded from the environment in the use of '$\mathcal{D}_{\Gamma,x:\sigma}[t]$'. In an implementation with De Bruijn environments, an explicit weakening would be used here.

  Recall from Fig. 5.2 that $\mathcal{D}[\Gamma, x : \sigma]_2 = \mathcal{D}[\Gamma]_2 \times \mathcal{D}[\sigma]_2(x)$, making the assignment '$\langle d_\Gamma, d_x\rangle = dy\ d$' well-typed.

- Units do not use variables and hence return a zero cotangent.
- A pair term has two subterms, thus its backpropagator add contributions from both subterms.
- Projections ('fst' and 'snd', in our language) discard values, hence their reverse derivative introduces a zero. Recall the discussion on the reverse derivative of sharing and dropping on page 34 for the more general picture.
- Coproduct injections profit from the dependent typing of $\mathcal{D}[-]_2$ to know that the received cotangent is compatible with the injection; recall Section 5.1.3.
- The $\mathcal{D}_\Gamma[\mathbf{case}]$ rule is relatively large because it combines dynamic control flow, variable binding and the presence of multiple subterms. Note that the applications $du\ d_x$ in the branches typecheck because under the equalities $u = \mathrm{inl}\ x$ and $u = \mathrm{inr}\ y$, which should be produced by the **case**-match, the $\mathcal{D}[\sigma \sqcup \tau]_2(u)$ in the type of du rewrites to, respectively, $\mathcal{D}[\sigma]_2(x)$ and $\mathcal{D}[\tau]_2(y)$. (Here it is paramount that $\mathcal{D}[-]_2$ is indexed on the primal, not on the original term.) Further note that although the bodies of the let-bindings inside the **case**-branches are lexically identical (modulo alpha-renaming), they cannot simply be moved outside the branches because their types differ.
- In $\mathcal{D}_\Gamma[\lambda x.\ t]$, first note that the binding for f is merely to avoid code duplication of the '$\mathcal{D}_{\Gamma,x:\sigma}[t]$' term; the right-hand side itself does not compute anything. This f is already almost of the correct type for the primal $\mathcal{D}[\sigma \to \tau]_1$; its backpropagator just needs to be composed with 'snd' to discard the environment cotangent contributions; this is done in f_p. At the bottom of the rule, the backpropagator for the lambda abstraction folds over the copower, re-running f on each pair and discarding the argument cotangents this time, leaving the environment cotangents, which are summed in copowfold and subsequently returned. Note that f_p has a name only to avoid having to repeat its definition in the typing of d.
- A function application has two subterms: the function and the argument; as with more basic rules like $\mathcal{D}_\Gamma[\langle s, t\rangle]$, the environment cotangent contributions come simply from invoking the backpropagators of both subterms on the appropriate arguments. In this case, df receives a log entry for this application (in a copower) and dx receives the cotangent with respect to the input as computed using the primal of the function subterm.
- Real constants are no different from units for differentiation.

- The sign function has a discrete output ($\mathbf{1} \sqcup \mathbf{1}$), and regardless of what 'sign x' evaluates to, $\mathcal{D}[\mathbf{1} \sqcup \mathbf{1}]_2(\text{sign } x)$ will rewrite to $\mathbf{1}$. A linear function sends zero to zero, and so does the backpropagator for 'sign'. Unusually for a CHAD rule, there is an unused variable: dx.

- In the general rule for scalar primitive operations, we need to be able to compute, somehow, the partial derivatives of the primitive operation in question. As we assume the primitive operations given in some set $\text{Op}_n^{\mathbb{R}}$, we also assume their partial derivatives given as $\partial_i \text{op}_{\mathbb{R}}(d; x_1, \ldots, x_n) = d \cdot \frac{\partial}{\partial x_i}(\text{op}_{\mathbb{R}}(x_1, \ldots, x_n))$. For example, for multiplication ($\text{op}_{\mathbb{R}} = (\cdot)$), we have $\partial_1 \text{op}_{\mathbb{R}}(d; x_1, x_2) = d \cdot x_2$ and $\partial_2 \text{op}_{\mathbb{R}}(d; x_1, x_2) = d \cdot x_1$, since $\frac{\partial(x \cdot y)}{\partial x} = y$ and $\frac{\partial(x \cdot y)}{\partial y} = x$.

- Integer constants are also no different from units for differentiation.

- Finally, integer primitive operations are inactive for differentiation. The backpropagators we get, as well as the backpropagator we must produce, are informationless: they can only take zero, and hence always produce zero.

Efficiency. There are a number of points where the naive transformation of this chapter can be critiqued on its practical efficiency.

- The derivative of function types is rather bad: the fact that every call to a lambda function leads to two evaluations of its backpropagator, means that an expression n lambdas deep may be differentiated 2^n times. This problem did not occur in the examples discussed in Section 5.1.5 because the only lambda function we differentiated there ($\lambda v.\ a \cdot v$) happens to be simple enough that the derivative with respect to its argument and the derivative with respect to its free variables do not share any computation.

  Even apart from this, creating all the copowers (lists) and appending them is hard to make very efficient on real hardware. Both of these issues are addressed in Sections 6.8 and 6.10.2, but a practically efficient implementation is future work.

- In various places in the transformation, we create zero cotangents (e.g. in $\mathcal{D}_\Gamma[r : \mathbb{R}]$) and one-hot cotangents (in $\mathcal{D}_\Gamma[x]$), and furthermore we add many cotangents together (e.g. in $\mathcal{D}_\Gamma[\langle s, t \rangle]$). These are not constant-time operations, and they can get very expensive in practice, especially if Γ contains large data types such as arrays. Addressing these issues is the main focus of Chapter 6.

- In $\mathcal{D}_\Gamma[t]$ for a term t, every subexpression of t has a corresponding backpropagator, and each of these backpropagators is a lambda function. While we have seen that these can mostly be optimised away by beta-reduction, this is not the case in the presence of dynamic control flow. As we have alluded to earlier, one can reformulate the code transformation to avoid creating lambda functions where none existed in the source program (see Section 7.3).

5.2 Differential geometry

To close this chapter, we provide the interested reader with some pointers on the strong link between CHAD and differential geometry. We also sketch how the categorical presentation of CHAD corresponds with the code transformation described in this chapter.

Given connected differentiable manifolds[18] M, N and a differentiable function $f : M \to N$, differential geometry writes the total derivative, or *pushforward*, of f at a point $x \in M$ as:

$$T_x f : T_x M \to T_{f\,x} N$$

For any $x \in M$, $T_x M$ is a vector space, the *tangent space* to M at x; notably, it is dependent on x. We leave the precise definition of $T_x M$ abstract, but its interpretation is the space of perturbations of x, or alternatively rates of change of x in M. We have $T_x \mathbb{R}^n = \mathbb{R}^n$.

A *cotangent space* $T^*_x M$ to M at x is just the dual vector space $(T_x M)^*$ of the corresponding tangent space, consisting of all linear forms (functionals) $\alpha : T_x M \multimap \mathbb{R}$. If the reverse derivative (the *pullback*) is to be a linear function between cotangent spaces, its direction is naturally flipped:

$$T^*_x f : T^*_{f\,x} N \to T^*_x M$$
$$T^*_x f\ \alpha = \alpha \circ T_x f$$

This identification of tangents with vectors and cotangents with dual vectors explains why some authors insist on writing tangents as *column* vectors and cotangents as *row* vectors (covectors). This convention also coincides with how forward derivatives and gradients appear in a Jacobian matrix (i.e. as columns and rows, respectively).

$T^*_x M$ naturally corresponds to $\mathcal{D}[M]_2(x)$ from the dependently-typed reverse-mode CHAD transformation described in Section 5.1, written $\overleftarrow{\mathcal{D}}$ in [Nunes

[18] Loosely speaking, an n-dimensional differentiable manifold is a patching-together of open subsets of $\mathbb{R}^n$ that overlap in a way that is "compatible" enough to enable differentiation in the manifold. $\mathbb{R}^n$ itself is a (trivial) example of a connected differentiable manifold. Some sources, including [Lee 2003], require infinite differentiability, but we need continuous first derivatives only.

and Vákár 2023, §8], and with non-dependent types (and excluding coproducts) in [Vákár and Smeding 2022, §7]. T naturally corresponds to the forward-mode $\vec{\mathcal{D}}$ in [Nunes and Vákár 2023, §8] / [Vákár and Smeding 2022, §7].

For more details on differential geometry and its treatment of tangent spaces, see any introductory text on the topic, e.g. [Lee 2003, ch. 3, ch. 11].

Category theory. From a categorical perspective, T can be seen to correspond to a functor:

$$T : \mathbf{Man} \to \mathbf{Fam}(\mathbf{CMon})$$

where **Man** denotes the category of connected differentiable manifolds, **CMon** the category of commutative monoids[19] and $\mathbf{Fam}(C)$ the *free coproduct completion* of C. An object of $\mathbf{Fam}(C)$ has the form $(A, (B_a)_{a\in A})$, where A is a set and B is a family of objects in C indexed by the set A.[20] A morphism $(A, (B_a)_{a\in A}) \to (C, (D_c)_{c\in C})$ consists of, suggestively named, a morphism $f_p : A \to C$ and an A-indexed set of morphisms $df_a : B_a \to D_{f_p\, a}$. Equivalently, this set of morphisms df_a can be seen as a single dependent function $df : (a : A) \to B_a \to D_{f_p\, a}$ (still writing Π-types using Agda syntax from page 212). We write the morphisms as pairs (f_p, df).

To be a category, we must have well-behaving identity and composition of morphisms. In $\mathbf{Fam}(C)$, these look as follows:

$$\begin{aligned}&\mathrm{id}_A = (a \mapsto a, a\ b \mapsto b)\\ &(g_p : B \to C, dg) \circ (f_p : A \to B, df) = (g_p \circ f_p, a\ b \mapsto dg\ (f_p\ a)\ (df\ a\ b))\end{aligned}$$

The reader may note that the right-hand side of the second line looks remarkably like the chain rule for forward derivatives (Eq. (2.15) on page 27). Because of this, it may be unsurprising that $\mathbf{Fam}(C)$ lends itself well for modelling derivatives. The definition of T capitalises on this observation by operating as follows on objects and morphisms. Given some differentiable manifold M, we define $T\ M$ as the pair $(|M|, (T_x\ M)_{x\in M})$ (in $\mathbf{Fam}(\mathbf{CMon})$), where $|M|$ is the plain set underlying the space M and $T_x\ M$ is the tangent space introduced for the pushforward on the previous page. A morphism (differentiable function) $f : M \to N$ is mapped under T to a pair of functions, as required: $f_p : |M| \to |N|$ (the primal function for f, given by $f_p = f$) and $df : (x : M) \to T_x\ M \to T_{f\,x}\ N$ (f's forward derivative: $df = Df$, using 'D' notation from Section 2.2).

To be a functor, T must preserve identity and function composition; it is easy to check that this holds using the chain rule.

[19] As noted on page 210, one may expect vector spaces here but CHAD ends up needing commutative monoids only.

[20] Under the standard interpretation where the objects in a category are the types in a language, $(A, (B_a)_{a\in A})$ corresponds to the dependent pair type $\Sigma_{a\in A}\ B_a$ for specific A and B.

Analogous to T, we also have a functor describing reverse derivatives:

$$T^* : \mathbf{Man} \to \mathbf{Fam}(\mathbf{CMon}^{\mathrm{op}})$$
$$T^*\, M = (|M|, (T^*_x\, M)_{x \in M})$$
$$T^*\, (f : M \to N) = (f : |M| \to |N|, (Df)^\top : (x : M) \to T^*_{f\,x}\, N \to T^*_x\, M)$$

There are two differences between T^* and T in this presentation:

1. The family of monoids that a space M is mapped to has, for each point $x \in M$, the cotangent space $T^*_x\, M$ instead of the tangent space $T_x\, M$.

2. The df part of $T^*\, f$ maps in the opposite direction between the monoids; this corresponds to the fact that T^* maps into **CMon**'s opposite category, $\mathbf{CMon}^{\mathrm{op}}$.

It may be instructive to check that T^* preserves identity and composition as required; the relevant properties of the reverse derivative are $(D(x \mapsto x))^\top\, x\, u = u$ and the reverse chain rule: $(D(g \circ f))^\top\, x\, u = (Df)^\top\, x\, ((Dg)^\top\, (f\, x)\, u)$.

Programming languages. One application of category theory is to model programming languages: categories describe objects and arrows between them; programming languages have types and functions between them. More precisely, we relate arrows in a category not to *functions* in a programming language, but to terms: a term $x_1 : \tau_1, \ldots, x_n : \tau_n \vdash t : \tau$ corresponds to an arrow $[\![\tau_1]\!] \times \cdots \times [\![\tau_n]\!] \to [\![\tau]\!]$ in a category, where $[\![\tau]\!]$ is the object corresponding to the type τ.[21] We can already see that for this to make sense, the category must have a notion of a product.

More generally, the structure available in a category limits the expressivity of languages it models. Connected differentiable manifolds have a well-defined product, so **Man** can express product types and functions between them, but coproduct types would correspond to manifolds with multiple connected components, and function types, being infinite-dimensional, are not easily reflected in the world of manifolds at all. Furthermore, since mathematical functions are total, languages that allow non-termination require the corresponding category to explicitly model partiality. All of this requires us to move from **Man** to more expressive categories; the theory of CHAD chooses to add much of this structure in freely-generated way. [Nunes and Vákár 2023]

Whichever category is chosen, T and T^* describe a transformation on objects and morphisms. Not all manifolds correspond to a sensible type in a programming language, and not all differentiable functions between manifolds are computable to begin with. Some are, however, and for those, T and T^* describe

[21] Identity arrows and composition respectively correspond to $x : \tau \vdash x : \tau$ and substitution.

objects and morphisms in the target category ($\mathbf{Fam}(\mathbf{CMon}^{(\mathrm{op})})$ if we stay with **Man**) that describe their (co)tangent space and derivative. CHAD observes that differentiation preserves structure well enough that these target objects and morphisms are again described by terms — now in a category of dependent pair types, or *containers* — and implements the resulting mapping from types to types ($T^* \llbracket \tau \rrbracket = \llbracket \Sigma_{x:\mathcal{D}[\tau]_1} \mathcal{D}[\tau]_2(x) \rrbracket$) and from terms to terms ($\mathcal{D}_\Gamma[t]$) with an explicit code transformation.

The code transformation that we described in this chapter is the one induced by T^*, with the caveat that we use the "extended" T^*, with as source category a more expressive choice than **Man** that has coproducts and exponentials (function types). These extensions do not exactly correspond to the differential geometry definitions we presented; for example, the x in $T^*_x M$ is an element of M, forcing us to take $|M|$ as the index set in $T^* M$, whereas we need a non-trivial mapping of *primals* (represented by $\mathcal{D}[-]_1$) to support function types.

The forward-differentiation functor T also yields a CHAD code transformation just like the one presented in this chapter, but since it separates the computation of the primal result and the forward derivative, it requires storing a tape of primals as if it was reverse AD. Thus, if one requires forward AD, one is better off with a method derived from dual-numbers forward AD.

6 Efficient CHAD: Complexity

In the previous chapter, we have presented CHAD as a code transformation for reverse AD with a number of desirable properties: it is mathematically principled with a proof of soundness, handles higher-order programs just fine, and has no runtime-interpreted tape like dual-numbers methods do (Chapters 3 and 4). However, as already briefly discussed in Section 5.1.6, CHAD has a number of efficiency issues that make it far too slow to use in practice. Furthermore, the fact that it produces dependently-typed code is rather inconvenient, as practical languages with a focus on performance tend to not support dependent types.

Some of the efficiency issues are severe enough that we can diagnose them by investigating the complexity of the transformation: the (complexity class of the) ratio of the time taken by the derivative program and the time taken by the original program, when executed on the same input. In Chapter 3, we considered this notion of complexity for dual-numbers reverse AD and fixed the issues we found; in this chapter, we do the same for CHAD. For the first-order fragment of the input language (i.e. without function types), we furthermore provide a formalised proof in Agda that we indeed fixed all complexity issues.

In order to do these things, we also reformulate the algorithm to produce simply-typed code, for ease of analysis as well as implementation. Finally, in the interest of future applicability to high-performance computing, we extend the algorithm to array programs in the sense of Section 2.1; in contrast to Chapter 4, where efficient support for second-order array operations incurred significant concessions to the generality of dual-numbers reverse AD, the array extension to CHAD is more promising.

This chapter is based on [Smeding and Vákár 2024] (published at POPL), and has a rewritten introduction and Section 6.1, a new observation in Section 6.3.2, some editorial changes for clarity and updated notation for consistency with the rest of the thesis.

Summary of contributions. Concretely, the contributions of this chapter are follows:

- We start by dropping support for functions (temporarily; re-added in Section 6.8) to focus on the CHAD's first-order core. We present the simply-typed version of CHAD with abstractified cotangent types (Section 6.1) and identify its complexity problems (Section 6.3). We solve these problems (up to log-factors) using two techniques:

  1. transparent reimplementation of the linear types in the target language in terms of sparse data types (Section 6.3.1);
  2. using state-passing style in a monad (Section 6.3.2), closely related to the Cayley transform of Section 3.5 on dual numbers.

- We show how these log-factors can be subsequently eliminated using (well-encapsulated) mutability (Section 6.4).

- We give a formalised complexity proof in Agda that the resulting implementation achieves the (optimal) computational complexity expected of reverse AD algorithms (Sections 6.3.3 and 6.5). This Agda formalisation has been extended by a student (in a master thesis [Meijs 2025] co-supervised by the author of this thesis) to additionally prove semantic correctness of the first-order CHAD algorithm, establishing full soundness and (complexity-)efficiency of first-order CHAD.

- We extend the first-order CHAD algorithm to array programs (Section 6.6), showing that the same techniques work. While we do not extend the formalised proof, we explain how the same complexity argument continues to work.

- Although we use mutable state, this state is only written to using (commutative, associative) accumulation operations, meaning that the algorithm can support parallel programs without much effort (Section 6.7). Significant caveats apply to the practical efficiency of the result, but we defer addressing them until Chapter 7.

- Finally, we return to function types: we identify the rather severe complexity problem in their derivative and fix it using defunctionalisation or closure conversion (Section 6.8), resulting in a complexity-efficient version of the full algorithm from Chapter 5.

Types:

$$\tau, \sigma \ ::= \ \mathbb{R} \mid \mathbb{Z} \mid \mathbf{1} \mid \sigma \times \tau \mid \sigma \sqcup \tau$$

Terms:

$$\begin{array}{rcll} t, s, r & ::= & x \mid \mathbf{let}\ x : \tau = s\ \mathbf{in}\ t \mid \langle\rangle \mid \langle s, t\rangle \mid \mathrm{fst}\ t \mid \mathrm{snd}\ t & \text{(vars; products)} \\ & \mid & \mathrm{inl}\ t \mid \mathrm{inr}\ t \mid \mathbf{case}\ s\ \mathbf{of}\ \{\ \mathrm{inl}\ x \to t_1 \mid \mathrm{inr}\ y \to t_2\ \} & \text{(coproducts)} \\ & \mid & r \mid \mathrm{sign}\ t \mid \mathrm{op}_{\mathbb{R}}(t_1, \ldots, t_n) & \text{(reals)} \\ & \mid & n \mid \mathrm{op}_{\mathbb{Z}}(t_1, \ldots, t_n) & \text{(integers)} \end{array}$$

Figure 6.1: The source language of this chapter's reverse AD transformation.

6.1 Basic reverse-mode CHAD

As stated in the introduction, for most of this chapter we focus on the first-order fragment of CHAD from Chapter 5, i.e. without function types. For completeness, the precise source language is given in Fig. 6.1; the typing is the same as in Chapter 5 (Fig. 5.1).

We use syntactic sugar like in Chapter 5: (z is a fresh variable, when applicable)

- $\mathbf{let}\ \langle x, y\rangle = s\ \mathbf{in}\ t \stackrel{\text{def}}{=} \mathbf{let}\ z = s\ \mathbf{in}\ \mathbf{let}\ x = \mathrm{fst}\ z\ \mathbf{in}\ \mathbf{let}\ y = \mathrm{snd}\ z\ \mathbf{in}\ t$
- $\lambda\langle x, y\rangle.\ t \stackrel{\text{def}}{=} \lambda z.\ \mathbf{let}\ \langle x, y\rangle = z\ \mathbf{in}\ t$
- $\mathbf{let}\ a = s_1; b = s_2\ \mathbf{in}\ t \stackrel{\text{def}}{=} \mathbf{let}\ a = s_1\ \mathbf{in}\ \mathbf{let}\ b = s_2\ \mathbf{in}\ t$

Given a well-typed program $\Gamma \vdash t : \tau$ of type τ in typing context $\Gamma = x_1 : \tau_1, \ldots, x_n : \tau_n$, we use the following as the (initial, naive) meta-type of reverse-mode CHAD:

$$\Gamma \vdash t : \tau \quad \rightsquigarrow \quad \mathcal{D}[\Gamma]_1 \vdash \mathcal{D}_\Gamma[t] : \mathcal{D}[\tau]_1 \times (\mathcal{D}[\tau]_2 \to \mathbf{EV}\ \mathcal{D}[\Gamma]_2) \tag{6.1}$$

Compared with Chapter 5, the dependent types have been lost, we stop writing the linear function arrow, and the tupling in $\mathcal{D}[\Gamma]_2$ has been moved to a separate definition **EV**. Thus, we simply have:

$$\mathcal{D}[x_1 : \tau_1, \ldots, x_n : \tau_n]_1 = x_1 : \mathcal{D}[\tau_1]_1, \ldots, x_n : \mathcal{D}[\tau_n]_1$$
$$\mathcal{D}[x_1 : \tau_1, \ldots, x_n : \tau_n]_2 = x_1 : \mathcal{D}[\tau_1]_2, \ldots, x_n : \mathcal{D}[\tau_n]_2$$

This $\mathbf{EV}\ \Gamma$ (short for "environment vector") is an abstract type that contains one value for each type τ_i in Γ. For now, we naively implement the abstract type as $\mathbf{EV}\ (x_1 : \tau_1, \ldots, x_n : \tau_n) = ((\mathbf{1} \times \tau_1) \times \cdots) \times \tau_n$, making Eq. (6.1) identical to (the simply-typed version of) the meta-type constructed in Chapter 5. We will later consider more efficient implementations of $\mathbf{EV}\ \Gamma$.

$$\mathcal{D}[\tau]_1 = \tau \quad \text{(true until Section 6.8)}$$
$$\mathcal{D}[\mathbf{1}]_2 = \underline{\mathbf{1}} \qquad \mathcal{D}[\mathbb{R}]_2 = \underline{\mathbb{R}} \qquad \mathcal{D}[\sigma \times \tau]_2 = \mathcal{D}[\sigma]_2 \mathbin{\underline{\times}} \mathcal{D}[\tau]_2$$
$$\mathcal{D}[\mathbb{Z}]_2 = \underline{\mathbf{1}} \qquad \mathcal{D}[\sigma \sqcup \tau]_2 = \mathcal{D}[\sigma]_2 \mathbin{\underline{\sqcup}} \mathcal{D}[\tau]_2$$

$$\begin{aligned}
\mathcal{D}_\Gamma[x : \tau] &= \langle x : \mathcal{D}[\tau]_2, \lambda d.\ \mathbf{one}_{x:\mathcal{D}[\tau]_2 \in \mathcal{D}[\Gamma]_2}\ d\rangle \\
\mathcal{D}_\Gamma[\mathbf{let}\ x : \tau = s\ \mathbf{in}\ t] &= \mathbf{let}\ \langle x, x'\rangle = \mathcal{D}_\Gamma[s]; \langle y, y'\rangle = \mathcal{D}_{\Gamma,x:\tau}[t] \\
&\quad\ \mathbf{in}\ \langle y, \lambda d.\ \mathbf{let}\ \langle d_1, d_2\rangle = \mathbf{split}_{\mathcal{D}[\Gamma]_2, \mathcal{D}[\tau]_2}(y'\ d) \\
&\qquad\qquad \mathbf{in}\ d_1 + x'\ d_2\rangle \\
\mathcal{D}_\Gamma[\langle\rangle] &= \langle\langle\rangle, \lambda\langle\rangle.\ \underline{0}\rangle \\
\mathcal{D}_\Gamma[\langle s, t\rangle] &= \mathbf{let}\ \langle x, x'\rangle = \mathcal{D}_\Gamma[s]; \langle y, y'\rangle = \mathcal{D}_\Gamma[t] \\
&\quad\ \mathbf{in}\ \langle\langle x, y\rangle, \lambda d.\ x'\ (\mathbf{lfst}\ d) + y'\ (\mathbf{lsnd}\ d)\rangle \\
\mathcal{D}_\Gamma[\mathrm{fst}\ t] &= \mathbf{let}\ \langle x, x'\rangle = \mathcal{D}_\Gamma[t]\ \mathbf{in}\ \langle \mathrm{fst}\ x, \lambda d.\ x'\ \langle\!|d, \underline{0}|\!\rangle\rangle \\
\mathcal{D}_\Gamma[\mathrm{snd}\ t] &= \mathbf{let}\ \langle x, x'\rangle = \mathcal{D}_\Gamma[t]\ \mathbf{in}\ \langle \mathrm{snd}\ x, \lambda d.\ x'\ \langle\!|\underline{0}, d|\!\rangle\rangle \\
\mathcal{D}_\Gamma[\mathrm{inl}\ t] &= \mathbf{let}\ \langle x, x'\rangle = \mathcal{D}_\Gamma[t]\ \mathbf{in}\ \langle \mathrm{inl}\ x, \lambda d.\ x'\ (\mathbf{lcastl}\ d)\rangle \\
\mathcal{D}_\Gamma[\mathrm{inr}\ t] &= \mathbf{let}\ \langle x, x'\rangle = \mathcal{D}_\Gamma[t]\ \mathbf{in}\ \langle \mathrm{inr}\ x, \lambda d.\ x'\ (\mathbf{lcastr}\ d)\rangle \\
\mathcal{D}_\Gamma\begin{bmatrix}\mathbf{case}\ s : \sigma \sqcup \tau\ \mathbf{of} \\ \quad \mathrm{inl}\ x \to t_1 \\ \quad \mathrm{inr}\ y \to t_2\end{bmatrix} &= \mathbf{let}\ \langle z, z'\rangle = \mathcal{D}_\Gamma[s]\ \mathbf{in} \\
&\quad\ \mathbf{case}\ z\ \mathbf{of} \\
&\qquad \mathrm{inl}\ x \to \mathbf{let}\ \langle x_1, x_2\rangle = \mathcal{D}_{\Gamma,x:\sigma}[t_1]\ \mathbf{in} \\
&\qquad\qquad \langle x_1, \lambda d.\ \mathbf{let}\ \langle v_1, v_2\rangle = \mathbf{split}_{\mathcal{D}[\Gamma]_2, \mathcal{D}[\sigma]_2}(x_2\ d) \\
&\qquad\qquad\qquad \mathbf{in}\ v_1 + z'\ (\mathbf{linl}\ v_2)\rangle \\
&\qquad \mathrm{inr}\ y \to \mathbf{let}\ \langle y_1, y_2\rangle = \mathcal{D}_{\Gamma,y:\tau}[t_2]\ \mathbf{in} \\
&\qquad\qquad \langle y_1, \lambda d.\ \mathbf{let}\ \langle w_1, w_2\rangle = \mathbf{split}_{\mathcal{D}[\Gamma]_2, \mathcal{D}[\tau]_2}(y_2\ d) \\
&\qquad\qquad\qquad \mathbf{in}\ w_1 + z'\ (\mathbf{linr}\ w_2)\rangle \\
\mathcal{D}_\Gamma[r] &= \langle r, \lambda_.\ \underline{0}\rangle \\
\mathcal{D}_\Gamma[\mathrm{sign}\ t] &= \langle \mathrm{sign}\ t, \lambda_.\ \underline{0}\rangle \\
\mathcal{D}_\Gamma[\mathrm{op}_\mathbb{R}(t_1, \ldots, t_n)] &= \mathbf{let}\ \langle x_1, x_1'\rangle = \mathcal{D}_\Gamma[t_1]; \ldots; \langle x_n, x_n'\rangle = \mathcal{D}_\Gamma[t_n] \\
&\quad\ \mathbf{in}\ \langle \mathrm{op}_\mathbb{R}(x_1, \ldots, x_n) \\
&\qquad ,\lambda d.\ x_1'\ (\partial_1 \mathrm{op}_\mathbb{R}(d; x_1, \ldots, x_n)) \\
&\qquad\qquad + \cdots + x_n'\ (\partial_n \mathrm{op}_\mathbb{R}(d; x_1, \ldots, x_n))\rangle \\
\mathcal{D}_\Gamma[n] &= \langle n, \lambda\langle\rangle.\ \underline{0}\rangle \\
\mathcal{D}_\Gamma[\mathrm{op}_\mathbb{Z}(t_1, \ldots, t_n)] &= \mathbf{let}\ \langle x_1, _\rangle = \mathcal{D}_\Gamma[t_1]; \ldots; \langle x_n, _\rangle = \mathcal{D}_\Gamma[t_n] \\
&\quad\ \mathbf{in}\ \langle \mathrm{op}_\mathbb{Z}(x_1, \ldots, x_n), \lambda\langle\rangle.\ \underline{0}\rangle
\end{aligned}$$

Figure 6.2: The basic reverse-mode CHAD definitions for transforming types and programs.

$$
\begin{array}{ll}
\mathbf{one}_{x:\tau\in\Gamma} : \tau \to \mathbf{EV}\ \Gamma & \partial_i \mathrm{op}_{\mathbb{R}} : \underline{\mathbb{R}} \to \mathbb{R}^n \to \underline{\mathbb{R}} \\
\mathbf{split}_{\Gamma,\tau} : \mathbf{EV}\ (\Gamma, x:\tau) \to \mathbf{EV}\ \Gamma \times \tau & (\text{for } \mathrm{op}_{\mathbb{R}} \in \mathrm{Op}^n_{\mathbb{R}},\ i \in \{1,\dots,n\}) \\
\mathbf{lfst} : \sigma \mathbin{\underline{\times}} \tau \to \sigma & \mathbf{lcastl} : \sigma \mathbin{\underline{\sqcup}} \tau \to \sigma \\
\mathbf{lsnd} : \sigma \mathbin{\underline{\times}} \tau \to \tau & \mathbf{lcastr} : \sigma \mathbin{\underline{\sqcup}} \tau \to \tau \\
\langle\!|-,-|\!\rangle : \sigma \to \tau \to \sigma \mathbin{\underline{\times}} \tau & \mathbf{linl} : \sigma \to \sigma \mathbin{\underline{\sqcup}} \tau \\
\underline{0}_\tau : \tau \qquad (+_\tau) : \tau \to \tau \to \tau & \mathbf{linr} : \tau \to \sigma \mathbin{\underline{\sqcup}} \tau
\end{array}
$$

Figure 6.3: The API for the linear types, which we view as abstract types that have multiple implementations.

For the special case where the input term is scalar-valued, i.e. $\Gamma \vdash t : \mathbb{R}$, the transformed term $\Gamma \vdash \mathcal{D}_\Gamma[t] : \mathbb{R} \times (\underline{\mathbb{R}} \to \mathbf{EV}\ \mathcal{D}[\Gamma]_2)$ computes, with sharing, the (primal) function value of t as well as a function that computes its *gradient* if we feed it 1 as an input. $\mathbf{EV}\ \mathcal{D}[\Gamma]_2$ then stores a representation of this gradient.

The CHAD algorithm that we use in this chapter is given in Fig. 6.2. Contrary to Chapter 5, we do not simply set $\mathcal{D}[\sigma \times \tau]_2 = \mathcal{D}[\sigma]_2 \times \mathcal{D}[\tau]_2$; instead, we have a separate set of *linear types*.[1] Specifically, relative to the source language (Fig. 6.1), we add the following types in the target language:

$$\tau, \sigma ::= \cdots \mid \sigma \to \tau \mid \underbrace{\underline{\mathbb{R}} \mid \underline{\mathbf{1}} \mid \sigma \mathbin{\underline{\times}} \tau \mid \sigma \mathbin{\underline{\sqcup}} \tau \mid \mathbf{EV}\ \Gamma}_{\text{linear types}}$$

The 'Γ' in '$\mathbf{EV}\ \Gamma$' must be an environment containing linear types only. We have corresponding additional terms:

$$
\begin{array}{lll}
t, s ::= \cdots \mid \lambda x : \tau.\ t \mid s\ t & & \text{(functions)} \\
\quad \mid \mathbf{one} \mid \mathbf{split} \mid \langle\!|\rangle\!| \mid \langle\!|s, t|\!\rangle \mid \mathbf{lfst} \mid \mathbf{lsnd} \mid \underline{0} \mid s + t & & \text{(linear types)} \\
\quad \mid \mathbf{lcastl} \mid \mathbf{lcastr} \mid \mathbf{linl} \mid \mathbf{linr} \mid \partial_i \mathrm{op}_{\mathbb{R}}(t_1, \dots, t_n) & & \text{(linear types)}
\end{array}
$$

with the typing of Fig. 6.3. In particular, all linear types are commutative monoids.

Figure 6.4 gives naive definitions of the linear types and primitives; later, we will replace these with more optimised ones. The linear operations $\partial_i \mathrm{op}_{\mathbb{R}} : \underline{\mathbb{R}} \to \mathbb{R}^n \to \underline{\mathbb{R}}$ are assumed to be implementations of the partial derivatives of the operations $\mathrm{op}_{\mathbb{R}}$, as in Chapter 5.

Finally, note that the **error** cases in Fig. 6.4 arise from the weakening to simple types: dependent typing in the style of Chapter 5 would prove (in the context of the full algorithm) that these **error** cases are unreachable. As stated, we use a

[1] Linear types in the sense that they come equipped with the algebraic structure of a commutative monoid and that the functions we consider between these types are monoid homomorphisms. Note, however, that we combine these types using the (bi)additive conjunction (the biproduct) rather than the multiplicative conjunction (the tensor product). In that sense, we can freely use the structural rules of dereliction and contraction. See [Vákár and Smeding 2022] for details.

$$\underline{\mathbb{R}} = \mathbb{R} \qquad \sigma \underline{\times} \tau = \sigma \times \tau \qquad \sigma \underline{\sqcup} \tau = \mathbf{1} \sqcup (\sigma \sqcup \tau)$$

$$\underline{0}_{\underline{\mathbb{R}}} = 0 \qquad \underline{0}_{\sigma \underline{\times} \tau} = \langle \underline{0}_\sigma, \underline{0}_\tau \rangle \qquad \underline{0}_{\sigma \underline{\sqcup} \tau} = \text{inl}\ \langle\rangle$$

$$s +_{\underline{\mathbb{R}}} t = s + t \qquad s +_{\sigma \underline{\times} \tau} t = \qquad s +_{\sigma \underline{\sqcup} \tau} t =$$

$$\quad \mathbf{let}\ \langle x_1, x_2 \rangle = s \qquad \quad \mathbf{case}\ t\ \mathbf{of}$$

$$\underline{\mathbf{1}} = \mathbf{1} \qquad\qquad \langle y_1, y_2 \rangle = t \qquad\qquad \text{inl}\ \_ \to s$$

$$\underline{0}_{\underline{\mathbf{1}}} = \langle\rangle \qquad\qquad \mathbf{in}\ \langle x_1 +_\sigma y_1, x_2 +_\tau y_2 \rangle \qquad\qquad \text{inr}\ (\text{inl}\ x) \to x_1 +_\sigma \mathbf{lcastl}\ t$$

$$s +_{\underline{\mathbf{1}}} t = \langle\rangle \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{inr}\ (\text{inr}\ x) \to x_2 +_\tau \mathbf{lcastr}\ t$$

$$\langle\!|-,-|\!\rangle = \lambda x.\ \lambda y.\ \langle x, y \rangle \qquad \mathbf{linl} = \lambda x.\ \text{inr}\ (\text{inl}\ x)$$

$$\mathbf{lfst} = \lambda x.\ \text{fst}\ x \qquad \mathbf{linr} = \lambda x.\ \text{inr}\ (\text{inr}\ x)$$

$$\mathbf{lsnd} = \lambda x.\ \text{snd}\ x \qquad \mathbf{lcastl} = \lambda x.\ \mathbf{case}\ x\ \mathbf{of}$$

$$\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{inl}\ \_ \to \underline{0}$$

$$\mathbf{one}_{x_i:\tau_i \in \Gamma} = \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{inr}\ (\text{inl}\ x) \to x$$

$$\quad \lambda x_i.\ \langle \underline{0}_{\tau_1}, \cdots, \underline{0}_{\tau_{i-1}}, x_i, \underline{0}_{\tau_{i+1}}, \cdots, \underline{0}_{\tau_n} \rangle \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{inr}\ (\text{inr}\ \_) \to \mathbf{error}$$

$$\mathbf{split}_{\Gamma,\tau} = \lambda y.\ y \qquad \mathbf{lcastr} = \lambda x.\ \mathbf{case}\ x\ \mathbf{of}$$

$$\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{inl}\ \_ \to \underline{0}$$

$$\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{inr}\ (\text{inl}\ \_) \to \mathbf{error}$$

$$\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{inr}\ (\text{inr}\ x) \to x$$

Figure 6.4: The naive implementation of the API required for our linear types that store cotangents. Nested pattern-matching desugars in the standard fashion to nested **case**. '**one**' produces a *one-hot* vector: an all-zero structure except for one element.

simply-typed presentation in this chapter to focus on operational concerns and to ease implementability in actual languages for numerical computation.

6.2 Key ideas

Key complexity criterion. In its basic form (without e.g. checkpointing), reverse AD should compute the gradient of a function $f : \mathbb{R}^n \to \mathbb{R}$ in time proportional to the time required to compute the original function. Put more precisely, there exist some (relatively small) constants c and c' such that for all functions $f : \mathbb{R}^n \to \mathbb{R}$ that are expressible in some programming language, the cost of computing the gradient $\nabla f(x)$ is less than c' plus c times the cost of computing $f(x)$, for any input x. It is important that c and c' are uniform: they depend neither on the program f nor on the input x. Formally, we therefore demand that:

$$\begin{aligned}&\exists c, c' > 0.\ \forall (x : \tau \vdash t : \mathbb{R}).\ \forall x : \tau.\ \forall d : \underline{\mathbb{R}}.\\ &\qquad \mathrm{cost}(\mathrm{snd}\ \mathcal{D}_\Gamma[t]\ d; x = x, d = d) \leq c' + c \cdot \mathrm{cost}(t; x = x)\end{aligned} \tag{6.2}$$

where $\mathrm{cost}(t; x_1 = v_1, \ldots, x_n = v_n)$ is the time cost of evaluating the program t in the environment where the variables $x_1, \ldots, x_n$ are in scope and have values $v_1, \ldots, v_n$. This is the key complexity property that we prove (in Section 6.5) that CHAD satisfies after the optimisations described in this chapter.[2] In fact, because of the inductive structure of the proof, we prove a stronger statement that generalises Eq. (6.2) to programs of the form $x_1 : \sigma_1, \ldots, x_n : \sigma_n \vdash t : \tau$.

Identifying complexity challenges. We want to prove this criterion by induction on the structure of programs. Therefore, Eq. (6.2) (generalised to terms of types other than $\mathbb{R}$) should apply not only to the input program as a whole, but also to all subterms. This is a very strong requirement! In particular, it immediately exposes efficiency problems with the naive implementation of the commutative monoids in Fig. 6.4: when we

1. discard values (e.g. with projections such as fst; see Fig. 6.2), we need zeros $\underline{0}_\tau$ to take constant time;

2. share values (use let-bound variables multiple times), we need cotangent addition $+_\tau$ to take constant time (seemingly — see below);

3. have terms with multiple subterms, such as $\langle -, - \rangle$, we need environment addition $+_{\mathbf{EV}\ \Gamma}$ to take constant time;

[2] To be precise, seeing as one of our optimisations makes CHAD generate monadic code, the complexity criterion will change slightly to include the handler **run** for the relevant monad.

4. reference variables, we need one-hot environment vectors $\mathbf{one}_{x:\tau\in\Gamma}$ to take constant time.

Solving the problems. Problems (1) and (4) we address in Section 6.3.1 by replacing the naive definitions of $\sigma \underline{\times} \tau$ and $\mathbf{EV}\ \Gamma$ with sparse versions. While these fixes keep the code transformation itself as-is, the tree-map we use to implement $\mathbf{EV}\ \Gamma$ introduces log-factors in the complexity of certain operations.

In Section 6.3.2, we then address problem (3) by eliminating environment vector additions entirely. We move from divide-and-conquer style (creating environment vectors in subterms and merging them where they meet in enclosing terms) to state-passing style. That is, we create a *single* environment vector at the beginning of the computation and update it with individual contributions as we go. This change puts the code in monadic style, meaning that we can even perform those updates *mutably* and eliminate the log-factors from certain operations again. At this point, due to the state-passing style, the only place where we still use $+_\tau$ is in the case for variables ($\mathcal{D}_\Gamma[x:\tau]$).

Proving the complexity criterion. It turns that the algorithm now already has the right complexity! Point (2) above is not *actually* a requirement, because we can use an *amortisation argument* to discount the additions performed in $\mathcal{D}_\Gamma[x:\tau]$ against the work done to build up the cotangents that are being added. The argument works by making two observations:

1. CHAD treats cotangents in an affine way (i.e. after adding something to a cotangent, the original cotangent is not used any more);

2. The addition of our sparsely represented cotangents can be performed in cost that is proportional to the size of their intersection, and the sum is *smaller* in size than the summands by precisely the size of this intersection.

Property (1) means that it is valid to associate a "computation budget" with each built-up cotangent value (this budget will not be magically duplicated), and property (2) means that $+_\tau$ preserves the invariant that we always have this budget. Section 6.3.3 explains the details.

We implement this amortisation argument by strengthening the induction hypothesis of the complexity proof to be aware of this computation budget. The theorem that we prove (Eq. (6.7) in Section 6.5) is thus slightly different from the original criterion, which fortunately follows as a corollary (Eq. (6.8)), yielding the final result. We have formalised the complexity proof in the Agda proof assistant. No encoding of big-O theory was necessary, because the only big-O expressions used in our development are of the form $f = O(g)$, which can be expressed simply by an existential constant, as indeed we did in Eq. (6.2).

Efficient CHAD for arrays. To show that CHAD, also in asymptotically-efficient form, extends to parallel array operations, Section 6.6 gives derivatives for three such operations:

- 'build s $(i.\ t)$', which constructs an array of length s with value t at position i (where i is a variable name and $\Gamma, i : \mathbb{Z} \vdash t : \tau$);
- $s\,!\,t$, which indexes the array s at index t;
- 'fold $(x.\ s)$ t', which reduces the non-empty array t : Array τ using the element combination function $\Gamma, x : \tau \times \tau \vdash s : \tau$.

The methodology applies also to further array operations that complete the full set supported by typical array languages. We have chosen this small subset to illustrate the most important and difficult points.

Arrays behave like product types in the sense that indexing behaves like a generalised projection. Consequently, it is natural to use the same sparsity tricks in $\mathcal{D}[\text{Array } \tau]_2$ as we used for $\mathcal{D}[\sigma \times \tau]_2$ — and more, because arrays can have any length and pairs always have length 2.[3] As we focus on complexity in this chapter, we choose to forgo practical efficiency on this point and define $\mathcal{D}[\text{Array } \tau]_2$ as 'Bag $(\mathbb{Z} \times \mathcal{D}[\tau]_2)$': a representation of a list of index–value pairs that supports constant-time concatenation. This data type works for the complexity property in a sequential setting, although we do not formally prove it in Agda. Of course, the practical efficiency and parallelisability of this kind of sparse array is rather bad, and indeed we rejected sparse arrays as a solution for the reverse derivative of array indexing in Chapter 4 (on page 148) for this reason; we discuss some mitigation approaches in Section 6.7, but only fix this properly in Chapter 7.

While it may seem that we gave up parallelisability already when converting the algorithm to monadic style, this is in fact not true: monads do not inherently prevent parallelism! The left-hand and right-hand side of a bind operation $(\gg\!=)$ are of course sequentialised, but such a bind operation only occurs in full generality in our output programs when there was already a data-dependency in the source program (such as for let-bindings; see Fig. 6.6 in Section 6.3.2). Such places indeed inherently do not have parallelism, but many others do: this possibility is evidenced by the use of command sequencing $x \gg y = x \gg\!= \lambda\_.\ y$ instead of the general $(\gg\!=)$.

Because we only use *accumulation* instead of fully general mutation in the monad — that is, we only *add* values to the state — we can rearrange effects at will (our monad is *commutative*) and thus parallelisation is possible, using atomic operations for the actual implementation. In particular, $(\gg)$ admits a parallel implementation.

[3] The fact that arrays are *variably* sized is not important here.

Closure conversion for efficient CHAD of lambdas. Naive CHAD of higher order functions leads to a duplication of work, due to separation of the transposed derivative w.r.t. function arguments and captured context variables. This inefficiency can be exploited to get an exponential blow-up in complexity. An obvious solution is to get rid of all captured context variables, i.e. to apply closure conversion before CHAD. One option for this is to apply full defunctionalisation first, which uses a global program analysis to reduce function types to a combination of finite sum types and tuples, constructs we already know how to differentiate. Another option, which does retain locality and compositionality of the code transformation, is to explain how to differentiate the infinite sum types or existential types required to do typed closure conversion. We discuss both in Section 6.8.

6.3 Finding and solving efficiency problems

The basic algorithm defined in Section 6.1 is relatively simple and pleasant to analyse. However, it fails to satisfy the complexity criterion in Eq. (6.2), so there are some complexity issues to address.

Identifying complexity problems. To spot the complexity problems in the algorithm from Fig. 6.2, we can try to prove the complexity criterion in Eq. (6.2) inductively. To do this, we first have to strengthen the statement to also apply to subexpressions with larger contexts and non-$\mathbb{R}$ result types:

$$\begin{aligned}&\exists c, c' > 0.\ \forall (x_1 : \sigma_1, \ldots, x_n : \sigma_n \vdash t : \tau).\ \forall x_1 : \sigma_1, \ldots, x_n : \sigma_n.\ \forall d : \mathcal{D}[\tau]_2.\\ &\quad \mathrm{cost}(\mathrm{snd}\ \mathcal{D}_\Gamma[t]\ d; x_1 = x_1, \ldots, x_n = x_n, d = d)\\ &\qquad \leq c' + c \cdot \mathrm{cost}(t; x_1 = x_1, \ldots, x_n = x_n)\end{aligned} \tag{6.3}$$

Interpreting Eq. (6.3) intuitively, the criterion states that the cost of evaluating $\mathcal{D}_\Gamma[t]$, plus the cost of calling the backpropagator in its second component, must be within a constant factor of the cost of evaluating the original expression t.[4]

For instance, consider $t = \langle t_1, t_2 \rangle$. We can see (in Fig. 6.2) that $\mathcal{D}_\Gamma[\langle t_1, t_2 \rangle]$ evaluates $\mathcal{D}_\Gamma[t_1]$ and $\mathcal{D}_\Gamma[t_2]$, and the body of the backpropagator calls the backpropagators of t_1 and t_2. By the induction hypothesis, these are offset by the evaluation of the original t_1 and t_2 on the left-hand side. Thus, the inductive step of the proof would hold for pair expressions — were it not for the (single) use of + on environment vectors $\mathbf{EV}\ \Gamma$ in the backpropagator: the fact that $+_{\mathbf{EV}\ \Gamma}$ is expensive on the naive representation of $\mathbf{EV}\ \Gamma$ from Section 6.1 makes the pair case problematic.

[4]The c' is convenient for the proof, but could technically be removed assuming that all terms have cost ≥ 1.

Type	Operation		Reason for appearance
EV Γ	zero	$\underline{0}_{\mathbf{EV}\ \Gamma}$	constants (e.g. $\langle\rangle$)
	plus	$+_{\mathbf{EV}\ \Gamma}$	multiple subterms (e.g. $\langle s, t\rangle$ or $\mathrm{op}_{\mathbb{R}}(t_1, \ldots, t_n)$)
	one-hots	$\mathbf{one}_{x:\tau\in\Gamma}$	variable references (x)
	splitting	$\mathbf{split}_{\Gamma,\tau}$	scopes (e.g. **let**, **case**)
Cotangents	zero	$\underline{0}_{\tau}$	unused variables
	plus	$+_{\tau}$	variable sharing; see Section 6.3.3
	one-hots	e.g. $\langle\!|d, \underline{0}|\!\rangle$	product projections (e.g. fst); already constant-time

Table 6.1: Operations required to be constant-time (seemingly, in the case of $+_\tau$).

Inefficient monoid structures on cotangents. Similar reasoning elsewhere in the code transformation uncovers some more operations that are required to be efficient and are potentially not. Naively (but see Section 6.3.3), all of the operations shown in Table 6.1 must be constant-time.

Because our source language type system (Fig. 6.1) does not have unbounded-size products (i.e. n-ary tuples), one-hot cotangents are not an issue, because just building the one-hot value from (constant-time) zeros will take bounded cost anyway. An example of this is the $\langle\!|d, \underline{0}|\!\rangle$ in $\mathcal{D}_\Gamma[\mathrm{fst}\ t]$ in Fig. 6.2. (Had we included n-ary tuples in our language, these one-hots would have been a larger issue, needing a solution similar to that for arrays (Section 6.6); after all, arrays differ from (homogeneous) n-ary tuples only in that their size is not known statically, which does not matter much from a complexity perspective.) Thus we can focus on the other operations.

As an example of the need for constant-time zero, consider the program t:

$$x : \mathbb{R} \times \tau \vdash \mathrm{op}_{\mathbb{R}}(\mathrm{fst}\ x, \ldots, \mathrm{fst}\ x) : \mathbb{R}$$

for some some n-ary operation $\mathrm{op}_{\mathbb{R}}$. Because of the $\underline{0}$ in the rule for $\mathcal{D}_\Gamma[\mathrm{fst}\ t]$, the backpropagator of the derivative of t — that is, $\mathrm{snd}\ \mathcal{D}_{x:\mathbb{R}\times\tau}[t]$ — will use $\underline{0}_{\mathcal{D}[\tau]_2}$ once for each occurrence of fst to create n zero values of type $\mathcal{D}[\tau]_2$. If $\underline{0}_{\mathcal{D}[\tau]_2}$ takes time T_0, then the backpropagator of the derivative of t will take time at least on the order of $n \cdot T_0$, whereas t itself is $O(n)$. Hence T_0 must indeed be constant, i.e. not dependent on τ.

In addition to the zeros, the $n-1$ uses of $+_{\mathcal{D}[\tau]_2}$ in $\mathrm{snd}\ \mathcal{D}_\Gamma[\mathrm{op}_{\mathbb{R}}(\ldots)]$ also take time, and thus it would seem that + needs to be constant-time as well, explaining its presence in the table above. However, as we explain in Section 6.3.3, with a smarter analysis we can weaken this requirement.

Our current, naive implementations of $\mathbf{EV}\ \Gamma$ and $\mathcal{D}[\tau]_2$ do not at all reach these constant-time requirements, so we need to do something about this. Indeed, so far, $\mathbf{EV}\ \Gamma$ is a simple product of the types in Γ (as defined in Section 6.1: $\mathbf{EV}\ (x_1 : \tau_1, \ldots, x_n : \tau_n) = ((\mathbf{1} \times \tau_1) \times \cdots) \times \tau_n$): this representation has expensive zero, plus and one-hot (all at least $O(n)$ work), although it supports efficient $\mathbf{split}_{\Gamma,\tau}$.

Solving the problems with sparse representations. We address the problem of expensive zeros in Section 6.3.1 by choosing *sparse* representations of the monoids that are not already sparse: $\mathcal{D}[\sigma \times \tau]_2$ and $\mathbf{EV}\ \Gamma$. Then (in Section 6.3.2) we lift the output of the transformation to monadic code in order to eliminate the need for $+_{\mathbf{EV}\ \Gamma}$, as well as to prepare for making **one** and **split** constant-time (which became logarithmic because of the sparse representation) using a mutable array in Section 6.4. Finally, we only have $+_\tau$ left, which then turns out to not be a problem any more: an amortisation argument (informally in Section 6.3.3, more formally in Section 6.5) shows that we can discount $+_{\mathcal{D}[\tau]_2}$ against building up of cotangent values.

6.3.1 Step 1: Sparsity

Sparse data structures aim to represent the uninteresting parts of a data structure as compactly as possible, focusing on the interesting (usually the *non-zero*) parts. Such representations not only conserve memory but also serve to avoid computing with many useless (zero) values, since the result is typically as uninteresting as the input: zero.

Fixing cotangent zeros. Given the tree structure of our types (which are built out of products and sums), a natural first attempt for a sparse representation is to add an explicit, redundant value representing "zero" for products. ($\underline{0}_{\mathbf{1}}$, $\underline{0}_{\mathbb{R}}$ and $\underline{0}_{\sigma \sqcup \tau}$ are already constant-time.) That is, we change the implementation of $\sigma \mathbin{\underline{\times}} \tau$ from $\sigma \times \tau$ to $\mathbf{1} \sqcup (\sigma \times \tau)$. The implementation of its API then changes to:

$$
\begin{array}{ll}
\sigma \mathbin{\underline{\times}} \tau = \mathbf{1} \sqcup (\sigma \times \tau) & \mathrm{inl}\ \_ +_{\sigma \mathbin{\underline{\times}} \tau} y = y \\
\underline{0}_{\sigma \mathbin{\underline{\times}} \tau} = \mathrm{inl}\ \langle\rangle & x +_{\sigma \mathbin{\underline{\times}} \tau} \mathrm{inl}\ \_ = x \\
\langle\!\langle x, y \rangle\!\rangle = \mathrm{inr}\ \langle x, y \rangle & \mathrm{inr}\ \langle x_1, x_2 \rangle +_{\sigma \mathbin{\underline{\times}} \tau} \mathrm{inr}\ \langle y_1, y_2 \rangle = \mathrm{inr}\ \langle x_1 + y_1, x_2 + y_2 \rangle
\end{array}
$$

$$
\mathbf{lfst}\ x = \mathbf{case}\ x\ \mathbf{of}\ \{\ \mathrm{inl}\ \_ \to \underline{0} \mid \mathrm{inr}\ x' \to \mathrm{fst}\ x'\ \}
$$

$$
\mathbf{lsnd}\ x = \mathbf{case}\ x\ \mathbf{of}\ \{\ \mathrm{inl}\ \_ \to \underline{0} \mid \mathrm{inr}\ x' \to \mathrm{snd}\ x'\ \}
$$

The result is that $\underline{0}_{\mathcal{D}[\tau]_2}$ is now constant-time for all types τ in our source-language type system (Fig. 6.1).

data EMap Γ — *all types in* Γ *must be monoids.*

empty	: EMap Γ	— $O(1)$
$get_{x:\tau\in\Gamma}$	: EMap $\Gamma \to \tau$	— $O(\log(\textit{map size}))$
$push_{\underline{0}}$	: EMap $\Gamma \to$ EMap $(\Gamma, x:\tau)$	— *only a type change;* $O(1)$
pop'	: EMap $(\Gamma, x:\tau) \to$ EMap Γ	— $O(\log(\textit{map size}))$*; map delete operation*
$modify_{x:\tau\in\Gamma}$	: $(\tau \to \tau) \to$ EMap $\Gamma \to$ EMap Γ	— $O(\log(\textit{map size}))$ + (*one call to the function*)
union	: EMap $\Gamma \to$ EMap $\Gamma \to$ EMap Γ	— *[see caption]; uses* + *on types in* Γ
pop	: EMap $(\Gamma, x:\tau) \to$ EMap $\Gamma \times \tau$	— *derived:* $pop\ e =$ **let** $y = e$ **in** $\langle pop'\ y, get_{x:\tau\in\Gamma}\ y\rangle$

Figure 6.5: The interface of the environment map, used in Section 6.3.1. Here, the notation "$x : \tau \in \Gamma$" means a pointer into Γ, i.e. an integer from 1 to the length of Γ. Ignoring type safety, this interface, including the noted time complexities, can be implemented with a standard immutable tree map. Regarding *union*: if passed two maps of size m and n, with $m \le n$, its runtime is $O(m \log(\frac{n}{m} + 1))$ + (*the required* + *calls*), and this complexity is optimal. [Brown and Tarjan 1979] The first term is $O(n)$ when $m \approx n$ and $O(\log(n))$ when m is small.

Fixing environment zeros and one-hots. Since we implemented **EV** $(x_1 : \tau_1, \ldots, x_n : \tau_n)$ as $((\mathbf{1} \times \tau_1) \times \cdots) \times \tau_n$ in Section 6.1, the $\underline{0}_{\mathbf{EV}\ \Gamma}$ in $\mathcal{D}_\Gamma[\langle\rangle]$ and the $\mathbf{one}_{x:\mathcal{D}[\tau]_2\in\mathcal{D}[\Gamma]_2}$ in $\mathcal{D}_\Gamma[x:\tau]$ still have to create a big tuple filled with zeros. A standard way to make a sequence (a total map from bounded integers to values) sparse is to use an associative array, often implemented as a balanced binary tree in programming languages[5] (which is essentially a *partial* map from keys to values). Since our keys are slightly odd — they are pointers into the environment, and the type of the associated value depends on the index — we will explicitly specify the interface of such a map data structure instead of directly instantiating the standard data structure. However, it should be clear that it can be implemented with a standard immutable tree-map, modulo type-safety. This API is shown in Fig. 6.5.

In the interface, *empty* creates an empty map, and $get_{x:\tau\in\Gamma}\ e$ gets the entry for x from the map e, returning $\underline{0}_\tau$ if there is no entry there yet. $push_{\underline{0}}$ and pop', respectively, extend and reduce the environment, where pop' performs actual work because it removes the now-extraneous entry from the underlying map

[5]And in purely functional programming languages, almost always, because there a *mutable* hashmap does not work well.

structure; $push_{\underline{0}}$ simply changes the type to make an additional key allowable, but does not need to add the key yet. (Conceptually, $push_{\underline{0}}$ pushes a $\underline{0}$, but zeros are elided due to sparsity.) $modify_{x:\tau\in\Gamma}\ f\ e$ applies the function f to the value in the map at the variable x, pre-initialising with $\underline{0}_\tau$ if necessary. *union* takes the union of the two maps, adding overlapping values using the addition operation $+_\tau$ of the monoids τ contained in Γ. Finally, *pop* is a derived operation.

Using this interface, we can implement $\mathbf{EV}\ \Gamma$ as $\mathrm{EMap}\ \Gamma$ and instantiate its methods as follows:

$$\begin{array}{ll} \underline{0}_{\mathbf{EV}\ \Gamma} = empty & \mathbf{one}_{x:\tau\in\Gamma}\ v = modify_{x:\tau\in\Gamma}\ (\lambda y.\ v + y)\ empty \\ +_{\mathbf{EV}\ \Gamma} = union & \mathbf{split}_{\Gamma,\tau} \quad = pop \end{array}$$

At this point, zero cotangents (due to our new sparse $\sigma \mathbin{\underline{\times}} \tau$) as well as zero and one-hot environment vectors can all be constructed in constant time as required. Note that the program transformation of Fig. 6.2 has not changed: simply the implementation of some of the types has changed.

Thus the remaining points of care in the derivative program are *addition* of cotangent values and environment cotangents, as well as $\mathbf{split}_{\Gamma,\tau}$, which has inadvertently become log-time with this change in representation of $\mathbf{EV}\ \Gamma$; we address these in the following two sections.

6.3.2 Step 2: Monadic lifting

Let us first focus on the addition operation on environment cotangents (with type $\mathbf{EV}\ \mathcal{D}[\Gamma]_2$), which occurs in the differentiated program whenever evaluation of the corresponding source program term involves evaluating more than one subterm.[6] Usually, however, these source program terms do not take time on the order of the size of the environment, and hence do not "pay" (in the sense of a direct inductive proof of the complexity criterion Eq. (6.3) as explained at the start of Section 6.3) for *union*, which is at least logarithmic in the size of its arguments.[7]

Fortunately, we have some yet-unexploited flexibility: because the environment cotangent is only ever modified by *adding* contributions to it (and since addition is commutative and associative), we can rearrange these additions at will. For example, instead of returning the environment cotangent contributions from each subprogram and merging the results using *union*, we can also pass a growing accumulator around in state-passing style, adding individual $\mathbf{one}_{x:\tau\in\Gamma}$

[6] In Fig. 6.2, this happens for let-bindings, pair constructors, case elimination and primitive operations (of arity ≥ 2).

[7] An amortisation argument in the style of Section 6.3.3 will not work here; Section 6.10.1 gives a counterexample.

contributions to it as the derivative program executes. Using a slightly modified definition of **one**:

$$\mathbf{one}'_{x:\tau\in\Gamma}\ v\ e \coloneqq \mathit{modify}_{x:\tau\in\Gamma}\ (\lambda y.\ v + y)\ e \tag{6.4}$$

we can add the single contribution (here v) from $\mathcal{D}_\Gamma[x:\tau]$ to the passed state e of type $\mathbf{EV}\ \Gamma$ in time $O(\log(\text{map size}))$ + (cost of '+').

To use this **one′**, we have to change the structure of the derivative program somewhat: instead of passing environment contributions upwards, merging them as control flows meet, we pass around a *single* environment cotangent in state-passing style, that we modify with **one′** each time we encounter a variable reference. The result is that the derivative program then lives inside a variant of a *local state monad* [Plotkin and Power 2002; Staton 2010]: it is a state monad[8] (that we will call **EVM**, for environment vector monad) whose state is divided up into components, one for each entry in the environment. $\underline{0}_{\mathbf{EV}\ \Gamma}$ becomes **return** $\langle\rangle$, $+_{\mathbf{EV}\ \Gamma}$ becomes $\gg$, $\mathbf{one}_{x:\tau\in\Gamma}$ becomes $\mathbf{one}'_{x:\tau\in\Gamma}$, and usages of $\mathbf{split}_{\Gamma,\tau}$ become usages of $\mathbf{scope}_{\Gamma,\tau}$ (see 'About the methods' below) instead, using $\gg\!=$ to extract the results:

$$\begin{aligned}
&\mathbf{EVM}\ \Gamma\ \tau = \mathrm{EMap}\ \Gamma \to \tau \times \mathrm{EMap}\ \Gamma && \text{— \textit{again all types in } } \Gamma \text{ \textit{must be monoids}}\\
&\mathbf{one}_{x:\tau\in\Gamma} : \tau \to \mathbf{EVM}\ \Gamma\ \mathbf{1} && \text{— \textit{implemented using } } \mathbf{one}'_{x:\tau\in\Gamma} \text{ \textit{(Eq. (6.4))}}\\
&\mathbf{scope}_{\Gamma,\tau} : \mathbf{EVM}\ (\Gamma, x:\tau)\ \sigma \to \mathbf{EVM}\ \Gamma\ (\sigma\times\tau) && \text{— \textit{initial } } \tau \text{ \textit{value is } } \underline{0}_\tau\\
&\mathbf{run} \quad : \mathbf{EVM}\ \Gamma\ \tau \to \overline{\Gamma} \to \tau \times \overline{\Gamma}
\end{aligned}$$

For **run**, we use the notation $\overline{x_1:\tau_1,\ldots,x_n:\tau_n} = ((\mathbf{1}\times\tau_1)\times\cdots)\times\tau_n$.

In contrast to the changes in Section 6.3.1, we now have to change the code transformation to produce monadic code; the updated code transformation is shown in Fig. 6.6.[9] It is helpful to compare the new typing of the code transformation (at the top of Fig. 6.6) with the original typing in Eq. (6.1), and to compare the new rules with Fig. 6.2. Note how the structure is the same.

About the methods. Notable here is that the role of **split** is now fulfilled by **scope**. In principle, because we are now state-passing, we must not only *pop* (split off) the outermost variable of the environment cotangent vector returned by the derivative of a subcomputation with an extended scope (e.g. the body of a **let**), but also *push* an empty entry on the incoming vector before being able to pass it on to that same subcomputation in the first place. Had we chosen to

[8] Due to the absence of a 'get' operation, it can also be seen as a (CPS-style) Writer monad. We discuss this in more detail under 'Writer monad' on page 247. To more explicitly describe its implementation, however, we will call it a state monad.

[9] Do-notation uses Haskell syntax: '**do** $x \leftarrow s; t$' means '$s \gg\!= \lambda x.\ t$', '**do** $s; t$' means '$s \gg \mathbf{do}\ t$'; otherwise '**do** t' means simply 't'. Newlines stand for ';'.

$$\Gamma \vdash t : \tau \quad \rightsquigarrow \quad \mathcal{D}[\Gamma]_1 \vdash \mathcal{D}_\Gamma[t] : \mathcal{D}[\tau]_1 \times (\mathcal{D}[\tau]_2 \to \mathbf{EVM}\ \mathcal{D}[\Gamma]_2\ \mathbf{1})$$

$$\begin{aligned}
\mathcal{D}_\Gamma[x : \tau] &= \langle x : \mathcal{D}[\tau]_1, \lambda d.\ \mathbf{one}_{x:\mathcal{D}[\tau]_2 \in \mathcal{D}[\Gamma]_2}\ d\rangle \\
\mathcal{D}_\Gamma[\mathbf{let}\ x : \tau = s\ \mathbf{in}\ t] &= \mathbf{let}\ \langle x, x'\rangle = \mathcal{D}_\Gamma[s]; \langle y, y'\rangle = \mathcal{D}_{\Gamma,x:\tau}[t] \\
&\quad \mathbf{in}\ \langle y, \lambda d.\ \mathbf{do}\ \langle\langle\rangle, dx\rangle \leftarrow \mathbf{scope}_{\mathcal{D}[\Gamma]_2,\mathcal{D}[\tau]_2}\ (y'\ d); x'\ dx\rangle \\
\mathcal{D}_\Gamma[\langle\rangle] &= \langle\langle\rangle, \lambda\langle\rangle.\ \mathbf{return}\ \langle\rangle\rangle \\
\mathcal{D}_\Gamma[\langle s, t\rangle] &= \mathbf{let}\ \langle x, x'\rangle = \mathcal{D}_\Gamma[s]; \langle y, y'\rangle = \mathcal{D}_\Gamma[t] \\
&\quad \mathbf{in}\ \langle\langle x, y\rangle, \lambda d.\ \mathbf{do}\ x'\ (\mathbf{lfst}\ d); y'\ (\mathbf{lsnd}\ d)\rangle \\
\mathcal{D}_\Gamma[\mathrm{fst}\ t] &= \mathbf{let}\ \langle x, x'\rangle = \mathcal{D}_\Gamma[t]\ \mathbf{in}\ \langle \mathrm{fst}\ x, \lambda d.\ x'\ (\!|d, \underline{0}|\!)\rangle \\
\mathcal{D}_\Gamma[\mathrm{snd}\ t] &= \mathbf{let}\ \langle x, x'\rangle = \mathcal{D}_\Gamma[t]\ \mathbf{in}\ \langle \mathrm{snd}\ x, \lambda d.\ x'\ (\!|\underline{0}, d|\!)\rangle \\
\mathcal{D}_\Gamma[\mathrm{inl}\ t] &= \mathbf{let}\ \langle x, x'\rangle = \mathcal{D}_\Gamma[t]\ \mathbf{in}\ \langle \mathrm{inl}\ x, \lambda d.\ x'\ (\mathbf{lcastl}\ d)\rangle \\
\mathcal{D}_\Gamma[\mathrm{inr}\ t] &= \mathbf{let}\ \langle x, x'\rangle = \mathcal{D}_\Gamma[t]\ \mathbf{in}\ \langle \mathrm{inr}\ x, \lambda d.\ x'\ (\mathbf{lcastr}\ d)\rangle \\
\mathcal{D}_\Gamma\begin{bmatrix}\mathbf{case}\ s : \sigma \sqcup \tau\ \mathbf{of} \\ \quad \mathrm{inl}\ x \to t_1 \\ \quad \mathrm{inr}\ y \to t_2\end{bmatrix} &= \mathbf{let}\ \langle z, z'\rangle = \mathcal{D}_\Gamma[s]\ \mathbf{in} \\
&\quad \mathbf{case}\ z\ \mathbf{of} \\
&\qquad \mathrm{inl}\ x \to \\
&\qquad\quad \mathbf{let}\ \langle x_1, x_2\rangle = \mathcal{D}_{\Gamma,x:\sigma}[t_1] \\
&\qquad\quad \mathbf{in}\ \langle x_1, \lambda d.\ \mathbf{do}\ \langle\langle\rangle, dz\rangle \leftarrow \mathbf{scope}_{\mathcal{D}[\Gamma]_2,\mathcal{D}[\sigma]_2}\ (x_2\ d) \\
&\qquad\qquad\qquad z'\ (\mathbf{linl}\ dz)\rangle \\
&\qquad \mathrm{inr}\ y \to \\
&\qquad\quad \mathbf{let}\ \langle y_1, y_2\rangle = \mathcal{D}_{\Gamma,y:\tau}[t_2] \\
&\qquad\quad \mathbf{in}\ \langle y_1, \lambda d.\ \mathbf{do}\ \langle\langle\rangle, dz\rangle \leftarrow \mathbf{scope}_{\mathcal{D}[\Gamma]_2,\mathcal{D}[\tau]_2}\ (y_2\ d) \\
&\qquad\qquad\qquad z'\ (\mathbf{linr}\ dz)\rangle \\
\mathcal{D}_\Gamma[r] &= \langle r, \lambda_.\ \mathbf{return}\ \langle\rangle\rangle \\
\mathcal{D}_\Gamma[\mathrm{sign}\ t] &= \langle \mathrm{sign}\ t, \lambda_.\ \mathbf{return}\ \langle\rangle\rangle \\
\mathcal{D}_\Gamma[\mathrm{op}_{\mathbb{R}}(t_1, \ldots, t_n)] &= \mathbf{let}\ \langle x_1, x_1'\rangle = \mathcal{D}_\Gamma[t_1]; \ldots; \langle x_n, x_n'\rangle = \mathcal{D}_\Gamma[t_n] \\
&\quad \mathbf{in}\ \langle \mathrm{op}_{\mathbb{R}}(x_1, \ldots, x_n) \\
&\qquad , \lambda d.\ \mathbf{do}\ x_1'\ (\partial_1 \mathrm{op}_{\mathbb{R}}(d; x_1, \ldots, x_n)) \\
&\qquad\qquad \vdots \\
&\qquad\qquad x_n'\ (\partial_n \mathrm{op}_{\mathbb{R}}(d; x_1, \ldots, x_n))\rangle \\
\mathcal{D}_\Gamma[n] &= \langle n, \lambda\langle\rangle.\ \mathbf{return}\ \langle\rangle\rangle \\
\mathcal{D}_\Gamma[\mathrm{op}_{\mathbb{Z}}(t_1, \ldots, t_n)] &= \mathbf{let}\ \langle x_1, _\rangle = \mathcal{D}_\Gamma[t_1]; \ldots; \langle x_n, _\rangle = \mathcal{D}_\Gamma[t_n] \\
&\quad \mathbf{in}\ \langle \mathrm{op}_{\mathbb{Z}}(x_1, \ldots, x_n), \lambda\langle\rangle.\ \mathbf{return}\ \langle\rangle\rangle
\end{aligned}$$

Figure 6.6: The CHAD definitions updated after Section 6.3.2.

use pure state-passing style (**EV** Γ = EMap $\Gamma \to$ EMap Γ), we would indeed have used $push_0$ from Fig. 6.5; however, here in monadic style, such separate **push** and **split** would not typecheck. Thus, **scope** combines both.

Fortunately, $\mathcal{D}_\Gamma$ is compositional, meaning that $\mathcal{D}_{\Gamma'}[s]$ is a subterm of $\mathcal{D}_\Gamma[t]$ whenever s is a subterm of t. (And $\mathcal{D}_\Gamma[t]$ does not depend in any other way on the structure of s.) Therefore, we can *scope* the usage of an extended environment to the monadic subcomputation that handles the subterm with that extended environment in the style of a local state monad. This scoping is done by the updated $\mathbf{scope}_{\Gamma,\tau}$ above: conceptually, it first extends the EMap Γ in the state to an EMap $(\Gamma, x : \tau)$ (the push step — semantically storing $\underline{0}_\tau$ in the new cell but operationally, because of sparsity, just changing the monad type), then runs the subcomputation of type **EVM** $(\Gamma, x : \tau)\ \sigma$ with that extended state, and finally pops off the extra value of type τ and returns it along with the return value of the subcomputation (of type σ).

Finally, **run** is the handler (see [Plotkin and Pretnar 2013]) of the monad.

Writer monad. Since there is no 'get' method in the interface of **EVM**, it behaves more like a writer monad than a state monad. This observation is very useful, because together with commutativity of the monoid, it implies that the monad is commutative: $a \gg b$ is semantically equivalent to $b \gg a$. This means that a and b could well be executed in parallel, if the updates in **one** and **scope** are done atomically. We explore parallelism of CHAD further in Section 6.7.

However, the observation can also be used to better semantically understand the monadically lifted algorithm. A traditional writer monad is a pair with a monoid: $\mathcal{M}\ \alpha = M \times \alpha$ for some monoid M, where:

$$\begin{aligned}
&\langle m_1, x\rangle \gg\!= f = \mathbf{let}\ \langle m_2, y\rangle = f\ x\ \mathbf{in}\ \langle m_1 + m_2, y\rangle\\
&\langle m_1, x\rangle \gg \langle m_2, y\rangle = \langle m_1 + m_2, y\rangle\\
&\mathbf{return}\ x = \langle 0, x\rangle
\end{aligned}$$

In particular, $\mathcal{M}\ \mathbf{1} = M \times \mathbf{1}$ is isomorphic to M. In most of the original transformation in Fig. 6.2 (excepting scoping constructs: **let** and **case**), environment cotangent values are simply produced (using **one** and $\underline{0}$) and combined (using +); if we replace $\mathcal{D}[\Gamma]_2$ by $\mathcal{D}[\Gamma]_2 \times \mathbf{1}$, most of the transformation can be trivially rewritten to run in a naive writer monad instead (mapping $\underline{0}$ to **return** $\langle\rangle$ and + to $\gg$), and the result is clearly semantically equivalent to the original. However, usages of **split** require peeking inside the monad to manipulate the monoid value.

This writer monad $\mathcal{M}\ \alpha$ can then be reimplemented, under the hood, as a (stronger) state monad $M \to M \times \alpha$, offering the same interface and still being clearly semantically equivalent to the original. **EVM** is simply this state monad but specialised to a particular family of monoids (EMap Γ), resulting in the possibility

of offering additional methods **scope** and **one** that work efficiently on this kind of state. Finally, the availability of **scope** means that **let** and **case** do not need to peek inside the monad implementation, ensuring that the monad is used black-box and can be reimplemented at will (as we do in Section 6.4).

What did we achieve? Let us look back at the operations in Table 6.1 and make up the balance. All zeros are now constant-time and $+_{\mathbf{EV}\ \Gamma}$ is gone. On the other hand, $\mathbf{one}_{x:\tau\in\Gamma}\ d$ adds (using $+_\tau$) the cotangent value d to the entry in the environment cotangent (in the monad state) corresponding to the variable x. This takes time logarithmic in the size of the environment cotangent (thus usually $O(\log|\Gamma|)$, unless most variables are unused), plus the time required to invoke $+_\tau$ on d and the running total. Furthermore, $\mathbf{split}_{\Gamma,\tau}$ is also logarithmic in the size of the environment cotangent.

In Section 6.4, we will replace the logarithmic-time operations with constant-time ones by using a mutable array instead of a persistent tree map (EMap); this will modify only the implementation of **EVM** and its methods, keeping the code transformation itself completely as-is.[10] Then, the only remaining potential problem is the cotangent addition ($+_\tau$) in **one**. However, as we discuss below in Section 6.3.3, we can amortise the cost of these additions against the *creation* of the cotangent values being added, meaning that the code transformation as it is now in Fig. 6.6 — with the efficient implementation of **EVM** from Section 6.4 — is actually *already finished and asymptotically efficient.*

6.3.3 Step 3: There is no step 3 (amortisation)

Affine use of cotangents. To see how we are going to argue that the use of $+_{\mathcal{D}[\tau]_2}$ in $\mathcal{D}_\Gamma[x:\tau]$ is already efficient after the monadic lifting of Section 6.3.2, first observe that in Fig. 6.6 (and already in Fig. 6.2), cotangent values are used in an *affine*[11] manner: they are mostly not duplicated, and when they are (in $\mathcal{D}_\Gamma[\langle s,t\rangle]$), the structure is split using **lfst** and **lsnd** before using the constituent parts affinely again. (We could encode this affine usage with a resource-aware type system, but this does not really bring benefit for our presentation; the observation is simply useful to explain why the amortisation argument will go through, and in a way it is proved by the amortisation argument itself.) Because of this affine usage, once a cotangent value has been added to another, only the sum will again be used elsewhere; the values used to build this sum will never be used again.

Addition of sparse structures. The second ingredient that we need is an observation about the cost of $+_\tau$. The addition of two sparse cotangent values

[10] We can do this because **EVM** is used as a black-box monad.

[11] That is: linear, but dropping is allowed.

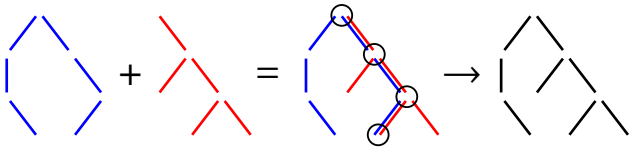


Figure 6.7: Pictured are two sparse structures, x in blue and y in red, together with their overlay. Leaves represent $\underline{\mathbb{R}}$, nodes with a vertical child indicate (one branch of) $\sigma \underline{\sqcup} \tau$, and nodes with diagonal children indicate $\sigma \underline{\times} \tau$. Omitted lines are children omitted due to sparsity. Computing the sum of x and y involves work on the overlap of the two structures; in this case, at 4 nodes. The structure of the sum is shown in black on the right.

x and y of type $\mathcal{D}[\tau]_2$, i.e. $x +_{\mathcal{D}[\tau]_2} y$, is essentially a zip of the two (sparse) structures. In this zip, we assume that the two cotangents, as far as they are present (i.e. not omitted due to sparsity), have equal structure; this is immediate for $\underline{\mathbf{1}}$, $\underline{\mathbb{R}}$ and $\sigma \underline{\times} \tau$, but a value of type $\sigma \underline{\sqcup} \tau$ can be both **linl** x and **linr** x. For $\sigma \underline{\sqcup} \tau$, we raise a runtime error if the two do not correspond (see Fig. 6.4 and the **error** calls inside **lcastl** and **lcastr** — this will not occur in practice because CHAD has been proved correct [Vákár and Smeding 2022]).

This zipping operation visits precisely the common "prefix", or rather the *intersection*, of both structures — no more, no less. Indeed, subtrees that occur in only one of the two arguments to + can simply be returned as-is, which is constant-time. An example is shown in Fig. 6.7. The circled nodes are the nodes that the two inputs share, i.e. neither has omitted. The implementation of + does not have to recurse into the left subtree of x (blue), nor into some of the branches of y (red).

More formally, define the size of a cotangent value as follows:

$$\begin{aligned}
&\mathrm{size}_\tau : \mathcal{D}[\tau]_2 \to \mathbb{Z}_{>0} \\
&\mathrm{size}_{\mathbf{1}}\ \langle\rangle = 1 && \mathrm{size}_{\sigma\underline{\sqcup}\tau}\ (\mathrm{inl}\ \langle\rangle) = 1 \\
&\mathrm{size}_{\mathbb{R}}\ x = 1 && \mathrm{size}_{\sigma\underline{\sqcup}\tau}\ (\mathrm{inr}\ (\mathrm{inl}\ x)) = 1 + \mathrm{size}_\sigma\ x \\
&\mathrm{size}_{\sigma\underline{\times}\tau}\ (\mathrm{inl}\ \langle\rangle) = 1 && \mathrm{size}_{\sigma\underline{\sqcup}\tau}\ (\mathrm{inr}\ (\mathrm{inr}\ y)) = 1 + \mathrm{size}_\tau\ y \\
&\mathrm{size}_{\sigma\underline{\times}\tau}\ (\mathrm{inr}\ \langle x, y\rangle) = 1 + \mathrm{size}_\sigma\ x + \mathrm{size}_\tau\ y
\end{aligned}$$

Then, formalising the observation of Fig. 6.7, we have the following inequality:

$$\begin{aligned}
\forall\tau.\ \forall a, b : \mathcal{D}[\tau]_2.\ \mathrm{cost}(x + y; x = a : \mathcal{D}[\tau]_2, y = b : \mathcal{D}[\tau]_2) \\
\leq c_\varphi \cdot (\mathrm{size}_\tau\ a + \mathrm{size}_\tau\ b - \mathrm{size}_\tau\ (a + b))
\end{aligned}$$

Here, c_φ is the number of computation steps (in our cost model) that + at most requires to fully handle one node in a cotangent value. Note that because the structure of $a+b$ is the union of the structures of a and b, the expression $(\mathrm{size}_\tau\ a + \mathrm{size}_\tau\ b - \mathrm{size}_\tau\ (a+b))$ is the size of the intersection of a and b. Now we define our

potential function as $\varphi_\tau\ d \coloneqq c_\varphi \cdot \mathrm{size}_\tau\ d$, measuring the number of computation steps that we budget in a node and can still use when consuming the cotangent value later. Then, rearranging terms, the inequality rewrites as follows:

$$\begin{aligned}&\forall \tau.\ \forall a, b : \mathcal{D}[\![\tau]\!]_2.\\ &\quad \mathrm{cost}(x + y; x = a : \mathcal{D}[\![\tau]\!]_2, y = b : \mathcal{D}[\![\tau]\!]_2) - \varphi_\tau\ a - \varphi_\tau\ b + \varphi_\tau\ (a + b) \leq 0\end{aligned} \tag{6.5}$$

in other words, after accounting for the potential flowing in (through a and b — these are computation steps we already counted elsewhere) and flowing out (through $a + b$ — the steps we are budgeting for later), addition is free.

Amortising sums. Armed with these two observations, first consider the simplified situation where a large number of cotangents are successively added to a single, threaded-through accumulator:

$$((((d + d_1) + d_2) + d_3) + d_4) + \ldots$$

Each of the additions involved takes time at most proportional to the size of the d_i added there — "size" being the number of materialised nodes in memory. Furthermore, since we do not duplicate structures, constructing d_i itself will also have taken time *at least* on the order of the size of d_i. Thus, this chain of additions (at most) duplicates the work done in constructing the d_i, which we had to do anyway; so essentially these additions are free. We have *amortised* the additions against the construction of their inputs.

Of course, in a general derivative program, the additions will not necessarily be done in such a linear fashion, but a more precise analysis of the above situation will show how the amortisation argument works in general. Indeed, because of our second observation (Fig. 6.7), the cost of $(\ldots) + d_i$ is not really proportional to the whole size of d_i: only the *intersection* of $(\ldots)$ and d_i is traversed in $+$. Furthermore, the parts of d_i that are *not* traversed are included as-is in the sum without processing. We can still amortise against those non-traversed subtrees of the sum!

Indeed, assuming we could still amortise against the entirety of both arguments d_1 and d_2 of an addition $d_1 + d_2$, their sum will consist of a number of subtrees that were included unchanged, connected by their intersection that $+$ did traverse once. However, on precisely that intersection, we had *two* input nodes we could amortise against: we can arbitrarily choose to amortise this addition against the overlapping part of d_1, and still have the corresponding part of d_2 to amortise against later. So in the end, the entire result of the addition can still be amortised against, preserving the invariant that we can always still amortise against the entirety of a cotangent value. And since we never copy cotangents, there is no risk of amortising against the same cotangent twice.

This affine usage is also the reason we can speak (informally) of potential being *carried by* a cotangent value, and thus flowing into, or out of, a computation together with that value.

It turns out that if we modify the full complexity criterion to account for potential flowing in and out like in Eq. (6.5), we get a statement that easily proves itself by induction; the only thing we still need to do then is implementing **EVM** efficiently, removing the logarithmic factors in its complexity. This optimisation is described in Section 6.4, after which Section 6.5 sketches the full complexity proof.

6.4 Getting rid of log-factors

The current implementation of **EVM** is not quite efficient enough for our purposes: $\mathbf{one}_{x\in\Gamma}\ d$ not only adds d (using $+$) to the value for x in the monad state, but also takes $O(\log|\Gamma|)$ time to find and update that value in the Map, a purely functional tree map. Similarly, **scope** has logarithmic overhead for updating values in the Map. These logarithmic computations violate the complexity criterion (because neither variable references nor let-bindings in the source program account for those logarithmic costs), but fortunately we can do better by replacing the Map with a mutable array.

Encoding mutability in a functional language can be done in multiple ways, and for the complexity proof it does not matter which we choose — as long as it can be encapsulated in the **EVM** API in such a way that its methods have the complexities listed in Table 6.2 (page 253).[12]

To understand the complexity of **run** that we require, note that its implementation has to allocate and deallocate an array, serialising the input and the output environments (of type $\overline{\Gamma}$) to and from that array; this is $O(|\Gamma|)$ work. We assume here that we are allowed to return cotangents in our sparse format; if not, the $|\Gamma|$ term in its complexity would increase to $\sum_{x:\tau\in\Gamma}$ size τ (where size τ is proportional to the time required to (de)serialise a value of type τ, and thus to convert back and forth to our sparse representation). The reason why the cost of **run** is not written as the slightly weaker $O(|\Gamma|)$ + (cost of m) is to allow some cancellation when computing with **run** in the proof: $O(|\Gamma|) - O(|\Gamma|)$ is not necessarily 0, but $c_{\text{run}} \cdot |\Gamma| - c_{\text{run}} \cdot |\Gamma| = 0$.

[12]Indeed, our Agda formalisation of the complexity proof is actually generic over all the implementations in this section, because it assumes only the complexities in Table 6.2 together with some equations on that API that define the semantics of the methods. The implementation is kept black-box; see Section 6.5.2.

Implementation. Assuming an implementation in Haskell, and assuming that we want to support the parallelisation discussed in Section 6.7 below, we need to resort to an implementation in terms of the `IO` monad. For illustration, a possible definition in GHC Haskell is as follows:[13]

$$\mathbf{EVM}\ \Gamma\ \alpha \coloneqq \texttt{Int} \to \texttt{IORef}\ (\texttt{IOVector}\ \texttt{Any}) \to \texttt{IO}\ \alpha$$

This is a reader monad, the reader context being the length of the environment in this subterm ($|\Gamma|$) together with a pointer (`IORef`) to a mutable vector (`IOVector`, from e.g. the `vector` package) of untyped values (`Any`); the values are untyped because we are implementing a mutable *heterogeneous* vector. Such a thing is not predefined in Haskell, so we must simulate it using a homogeneous vector of untyped values to and from which we `unsafeCoerce`. We use a pointer to the vector because in the **scope** method, if the vector is not large enough for the extended environment, we have to reallocate the array.[14]

In the non-parallel context, `ST` [Launchbury and Jones 1994] is sufficient as a replacement for `IO`. Meanwhile, in an imperative language such as OCaml, the required (atomic, in the case of parallelism) mutability is already present everywhere, meaning that $\mathbf{EVM}\ \Gamma\ \tau \coloneqq \texttt{Int} \to \text{Array}\ \texttt{Any} \to \tau$ suffices, with `Any` standing for the uninformative type, such as `Object` or `void*`.

Let us briefly look at some alternative implementations. To get a more functional-style implementation (but unfortunately not parallel!), one can use resource-linear types (such as those implemented in Linear Haskell [Bernardy et al. 2018]). In this case, the monad looks as follows:

$$\mathbf{EVM}\ \Gamma\ \alpha \coloneqq \texttt{Int} \to \text{EArr}\ \Gamma \multimap\ !\alpha \times \text{EArr}\ \Gamma$$

where EArr Γ is a mutable array with methods analogous to those of EMap Γ. The changes are the usual ones for a resource-linear mutable array (giving back the input array in the result and changing some arrows to resource-linear ones); for an example, see the mutable arrays in the `linear-base` package.[15] $!\tau$ denotes an unrestricted type, e.g. `Ur` τ in Linear Haskell.

If the reader is unfamiliar with resource-linear types, a simple intuitive stand-in is to use the original, purely functional State monad ($\mathbf{EVM}\ \Gamma\ \tau \coloneqq \text{EMap}\ \Gamma \to \tau \times \text{EMap}\ \Gamma$), but assume that EMap somehow has a magical implementation where *pop*′, $\mathit{modify}_{x:\tau\in\Gamma}\ f$ and $\mathit{get}_{x:\tau\in\Gamma}$ from Fig. 6.5 all run in $O(1)$ (apart from the cost of calling f once in *modify*).

[13] For a full implementation of this monad in GHC Haskell, see Section 6.10.5.

[14] If a growing array resizes to twice its size each time the underlying buffer is exhausted, the total amount of reallocation and copying work is linear in the final array length (as $\sum_{i=0}^{\lceil \log_2 n \rceil} 2^i = 2^{\lceil \log_2 n \rceil + 1} - 1 \leq 2n = O(n)$), and can thus be amortised away.

[15] `https://hackage.haskell.org/package/linear-base-0.5.0/docs/Data-Array-Mutable-Linear.html`

Method	Cost
$\mathbf{one}_{x:\tau\in\Gamma} : \tau \to \mathbf{EVM}\ \Gamma\ \mathbf{1}$ $\mathbf{one}_{x:\tau\in\Gamma}\ (y : \tau) = (...)$	$O(1)$ + (cost of adding y to the value for x in the array)
$\mathbf{scope}_{\Gamma,\tau} : \mathbf{EVM}\ (\Gamma, x : \tau)\ \sigma \to \mathbf{EVM}\ \Gamma\ (\sigma \times \tau)$ $\mathbf{scope}_{\Gamma,\tau}\ (m : \mathbf{EVM}\ (\Gamma, x : \tau)\ \sigma) = (...)$	$O(1)$ + (cost of m)
$\mathbf{run} : \mathbf{EVM}\ \Gamma\ \tau \to \overline{\Gamma} \to \tau \times \overline{\Gamma}$ $\mathbf{run}\ (m : \mathbf{EVM}\ \Gamma\ \tau)\ (env : \overline{\Gamma}) = (...)$	$O(1) + c_{\text{run}} \cdot \lvert\Gamma\rvert$ + (cost of m)

Table 6.2: The API of $\mathbf{EVM}\ \Gamma\ \tau$ with the complexities that we assume.

In the proof sketch in the following section, we will assume the complexities in Table 6.2, plus some properties about the semantics of these methods; these semantical properties are satisfied by all implementations discussed in this section, and will be left unstated here but are formulated precisely in the Agda formalisation in `spec/LACM.agda` (see Section 6.5.2).

6.5 Complexity proof

Because the derivative program now lives in a monad, the (generalised) complexity criterion (Eq. (6.3)) has to be modified to include its handler, **run**:

$$\begin{aligned}
&\exists c, c' > 0.\ \forall (x_1 : \sigma_1, \ldots, x_n : \sigma_n \vdash t : \tau).\ \forall x_1 : \sigma_1, \ldots, x_n : \sigma_n.\ \forall d : \mathcal{D}[\tau]_2.\\
&\forall d_1 : \mathcal{D}[\sigma_1]_2, \ldots, d_n : \mathcal{D}[\sigma_n]_2.\\
&\quad \text{cost}(\mathbf{run}\ (\text{snd}\ \mathcal{D}_\Gamma[t]\ d)\ env_0; \\
&\qquad x_1 = x_1, \ldots, x_n = x_n, d = d, env_0 = \langle d_1, \ldots, d_n \rangle) \\
&\quad \le c' + c \cdot \text{cost}(t; x_1 = x_1, \ldots, x_n = x_n) + c_{\text{run}} \cdot n
\end{aligned} \tag{6.6}$$

Since **run** takes an initial environment cotangent (to be accumulated into) as an additional argument, we need to provide one (env_0); we generalise over which environment the monad is initialised with to enable a proof by induction. The constant c_{run} is from Table 6.2 (see below for a discussion).

The next step is to account for amortisation, by subtracting incoming potential from the cost of the scrutinised expression (i.e. making incoming potential available as free computation steps), and adding outgoing potential to its cost (i.e. declaring outgoing potential as additional computation steps). This yields the

following criterion: (the same as Eq. (6.6) except for the highlighted fifth line)

$$\begin{aligned}
&\exists c, c' > 0.\ \forall (x_1 : \sigma_1, \ldots, x_n : \sigma_n \vdash t : \tau).\ \forall x_1 : \sigma_1, \ldots, x_n : \sigma_n.\ \forall d : \mathcal{D}[\tau]_2.\\
&\forall d_1 : \mathcal{D}[\sigma_1]_2, \ldots, d_n : \mathcal{D}[\sigma_n]_2.\\
&\quad \mathrm{cost}(\mathbf{run}\ (\mathrm{snd}\ \mathcal{D}_\Gamma[t]\ d)\ env_0;\\
&\qquad\quad x_1 = x_1, \ldots, x_n = x_n, d = d, env_0 = \langle\langle\langle\rangle, d_1\rangle, \ldots\rangle, d_n\rangle)\\
&\quad - \varphi\ d - \textstyle\sum_{i=1}^n \varphi\ d_i + \sum_{i=1}^n \varphi\ res_i\\
&\qquad \leq c' + c \cdot \mathrm{cost}(t; x_1 = x_1, \ldots, x_n = x_n) + c_{\mathrm{run}} \cdot n
\end{aligned} \tag{6.7}$$

where we have abbreviated using res_i the values that would be bound in:

$$\langle \_, \langle\langle\langle\rangle, res_1\rangle, \ldots\rangle, res_n\rangle\rangle = \mathbf{run}\ (\mathrm{snd}\ \mathcal{D}_\Gamma[t]\ d)\ \langle\langle\langle\rangle, d_1\rangle, \ldots\rangle, d_n\rangle$$

We prove Eq. (6.7) by induction on t; Section 6.5.1 gives a proof sketch illustrating the main ideas.

From Eq. (6.7) we derive a corollary that more directly states what a user can expect from our optimised version of CHAD. We initialise env_0 with a tuple of zeros (i.e. $d_i = \underline{0}_{\sigma_i}$), because the gradient program computes gradient + env_0 and we want just the gradient; we also move the construction of env_0 into the term on the left-hand side, increasing its work by $O(n)$ (captured in c'' below). Then we rearrange terms, substitute $\varphi\ d = c_\varphi \cdot \mathrm{size}\ d$, and weaken by forgetting the $\sum_{i=1}^n \varphi\ res_i$ term. This eliminates φ from the theorem statement:

$$\begin{aligned}
&\exists c, c', c'' > 0.\ \forall (x_1 : \sigma_1, \ldots, x_n : \sigma_n \vdash t : \tau).\ \forall x_1 : \sigma_1, \ldots, x_n : \sigma_n.\ \forall d : \mathcal{D}[\tau]_2.\\
&\quad \mathrm{cost}(\mathbf{run}\ (\mathrm{snd}\ \mathcal{D}_\Gamma[t]\ d)\ \langle\langle\langle\rangle, \underline{0}_{\sigma_1}\rangle, \ldots\rangle, \underline{0}_{\sigma_n}\rangle;\\
&\qquad\quad x_1 = x_1, \ldots, x_n = x_n, d = d)\\
&\qquad \leq c' + c \cdot \mathrm{cost}(t; x_1 = x_1, \ldots, x_n = x_n) + c'' \cdot n + c_\varphi \cdot \mathrm{size}\ d
\end{aligned} \tag{6.8}$$

Note that $c'' \cdot n$ captures three things: $c_{\mathrm{run}} \cdot n$, the $\sum_{i=1}^n \varphi\ d_i$ term and the $O(n)$ creation of the zero tuple $\langle\langle\langle\rangle, \underline{0}_{\sigma_1}\rangle, \ldots\rangle, \underline{0}_{\sigma_n}\rangle$.

This is our final complexity theorem, and our Agda formalisation proves Eq. (6.8).[16] Note that Eq. (6.8) is the same as Eq. (6.3) apart the additional $c'' \cdot n +$ size d term on the right-hand side (as well as inserting the call to **run**). However, this term is usually small, because even if a program has many inputs, these are typically organised in data structures rather than being passed as a large set of n separate inputs; furthermore, for most applications, we feed reverse-mode differentiated code very sparse cotangents d (e.g. basis vectors) for which 'size d' is small. And it is actually unsurprising: realistically, any reverse AD algorithm will need some setup work per argument, if only allocating an array for them, and the incoming cotangent d is typically going to need some processing. Our mentioning it explicitly here is just to be faithful to our formalised proof.

[16] The constants used there are $c = 34$, $c' = 5$ and $c'' = 4$; it assumes for simplicity that $c_\varphi = 1$. We stress that this constant 34 is not meaningful in practice because our cost model does not distinguish between various constant-time operations.

Cost model. So far, we have left the cost model of our complexity proof implicit, but to write such a proof one of course has to have a fixed cost model. Section 6.10.3 has a full definition, as does of course the Agda formalisation (see Section 6.5.2), but it can be briefly summarised as follows:

- The model assumes standard call-by-value semantics.
- The model is very conservative: all computations that could cost time are given non-zero cost. For example, **let** $x = s$ **in** t is given cost $1 + (\text{cost of } s) + (\text{cost of } t)$: we consider variable binding to be a potentially costly operation. Furthermore, the expression $\lambda x.\ t$ costs 1 plus the number of free variables of t: we count the allocation of the closure as potentially costly.
- Simultaneously, we do not care about the relative cost of various constant-time operations. Scalar multiplication has cost 1, as does allocating a fixed amount of memory. These operations are not at all comparable in their practical runtime, but we consider them both constant-time. This is because the proof is only about asymptotic complexity, not about absolute runtime.

6.5.1 Proof sketch: Induction on t

The statement being proved here is Eq. (6.7). We consider three representative cases of terms t in the induction: (1) the very simplest case of $t = \langle\rangle$ (where we do not require an induction hypothesis), to illustrate some of the basic book-keeping about potential flowing in and out of computations; (2) a slightly more complex case of $t = \langle t_1, t_2\rangle$ to show how we invoke the induction hypothesis; and (3) the case of variable references $t = x$, which is the only case in the whole proof where real work happens as this is where the potential is actually used, so we cannot merely cancel out the incoming and outgoing potentials.

The proof sketch uses "$O(1)$" as notation to indicate some unspecified bounded value whose bounds are independent of any of the other variables in the proof. We use this to abbreviate the cost of some constant-time work whose exact cost is immaterial to the argument it appears in.

Simple case. Let us first consider the simplest case: $t = \langle\rangle$ in the environment $x_1 : \tau_1, \ldots, x_n : \tau_n$; its transformation rule can be found in Fig. 6.6. The expression **run** $(\text{snd}\ \langle\langle\rangle, \lambda d.\ \textbf{return}\ \langle\rangle\rangle\ d)\ env_0$, which Eq. (6.7) scrutinises for this t, evaluates in $O(1) + c_{\text{run}} \cdot n$ steps ($c_{\text{run}} \cdot n$ are necessary to serialise and deserialise env_0) to $\langle\langle\rangle, env_0\rangle$. Further, $res_i = (env_0)_i = d_i$ and thus $-\sum_{i=1}^{n} \varphi\ d_i + \sum_{i=1}^{n} \varphi\ res_i = 0$. Noting that $\varphi\ d = 1$ for all $d : \underline{\mathbf{1}}$, Eq. (6.7) simplifies to the following clearly true inequality:

$$\exists c, c' > 0.\ O(1) + c_{\text{run}} \cdot n - 1 \leq c' + c + c_{\text{run}} \cdot n.$$

Note what happened with the potential: the first source of incoming potential (in $d : \mathcal{D}[\mathbf{1}]_2$) was ignored, resulting in an extra -1 term on the left-hand side of the inequality. This only made proving the theorem "easier": we got some unused free computation steps. The second source of incoming potential (in $d_1, \ldots, d_n$) cancelled exactly against the outgoing potential (in $res_1, \ldots, res_n$) because we did not change the accumulated environment cotangent.

Subterms and sparsity. Now let us consider the term $t = \langle t_1, t_2 \rangle$. When we evaluate the scrutinised expression, **run** (snd $\mathcal{D}_\Gamma[t]$ d) env_0, we are going to do the following things in addition to some constant-cost work (see Fig. 6.6):

1. Evaluate $\mathcal{D}_\Gamma[t_1]$;
2. Evaluate $\mathcal{D}_\Gamma[t_2]$;
3. Call snd $\mathcal{D}_\Gamma[t_1]$ on the argument **lfst** d;
4. Call snd $\mathcal{D}_\Gamma[t_2]$ on the argument **lsnd** d;
5. Sequence the results of (3.) and (4.) in the monad and **run** the result.

In short, this is equivalent (apart from some constant-cost work) to evaluating the expression:

$$\mathbf{run}\ (\mathrm{snd}\ \mathcal{D}_\Gamma[t_1]\ (\mathbf{lfst}\ d) \gg \mathrm{snd}\ \mathcal{D}_\Gamma[t_2]\ (\mathbf{lsnd}\ d))\ env_0$$

Because of the implementation as a state monad, we have (if we define $\langle v, env' \rangle \coloneqq \mathbf{run}\ a\ env$):

$$\begin{aligned} &\mathrm{cost}(\mathbf{run}\ (a \gg\!= \lambda x.\ b)\ e; e = env, \Gamma) \\ &\quad = O(1) + \mathrm{cost}(\mathbf{run}\ a\ e; e = env, \Gamma) \\ &\qquad + \mathrm{cost}(\mathbf{run}\ b\ e; e = env', x = v, \Gamma) - c_{\mathrm{run}} \cdot |\Gamma| \end{aligned} \tag{6.9}$$

where the term $-c_{\mathrm{run}} \cdot |\Gamma|$ arises because in the left-hand side, we (de)serialise env only once whereas in the right-hand side we do so twice: we have to subtract one of the two to make the left and right-hand sides equal. Now define for convenience in the below:

$$\begin{aligned} \langle \_, env' \rangle &\coloneqq \mathbf{run}\ (\mathrm{snd}\ \mathcal{D}_\Gamma[t_1]\ (\mathbf{lfst}\ d))\ \langle \langle \langle \rangle, d_1 \rangle, \ldots \rangle, d_n \rangle \\ \langle \_, env'' \rangle &\coloneqq \mathbf{run}\ (\mathrm{snd}\ \mathcal{D}_\Gamma[t_2]\ (\mathbf{lsnd}\ d))\ env' \end{aligned}$$

Using the above lemma about the cost of bind (Eq. (6.9)) in simplifying Eq. (6.7), we get the following:

$$\begin{aligned}
&\exists c, c' > 0.\ \forall (x_1 : \sigma_1, \ldots, x_n : \sigma_n \vdash t : \tau).\ \forall x_1 : \sigma_1, \ldots, x_n : \sigma_n.\ \forall d : \mathcal{D}[\tau]_2.\\
&\forall d_1 : \mathcal{D}[\sigma_1]_2, \ldots, d_n : \mathcal{D}[.]_{\sigma_n}\\
&\quad O(1) + \mathrm{cost}(\mathbf{run}\ (\mathcal{D}_\Gamma[t_1]\ d)\ env_0;\\
&\qquad\qquad x_1 = x_1, \ldots, x_n = x_n, d = \mathbf{lfst}\ d, env_0 = \langle\langle\langle\rangle, d_1\rangle, \ldots\rangle, d_n\rangle)\\
&\qquad + \mathrm{cost}(\mathbf{run}\ (\mathcal{D}_\Gamma[t_2]\ d)\ env;\\
&\qquad\qquad x_1 = x_1, \ldots, x_n = x_n, d = \mathbf{lsnd}\ d, env = env')\\
&\qquad - c_{\mathrm{run}} \cdot n - \varphi\ d - \textstyle\sum_{i=1}^n \varphi\ d_i + \sum_{i=1}^n \varphi\ res_i\\
&\quad \leq c \cdot \mathrm{cost}(\langle t_1, t_2\rangle; x_1 = x_1, \ldots, x_n = x_n) + c' + c_{\mathrm{run}} \cdot n
\end{aligned} \tag{6.10}$$

Now the two big 'cost' calls match the ones in the induction hypotheses for t_1 and t_2 (Eq. (6.7)); adding the two induction hypotheses together we get the following proposition:

$$\begin{aligned}
&\mathrm{cost}(\mathbf{run}\ (\mathrm{snd}\ \mathcal{D}_\Gamma[t_1]\ d)\ env_0;\\
&\qquad x_1 = x_1, \ldots, x_n = x_n, d = \mathbf{lfst}\ d, env_0 = \langle d_1, \ldots, d_n\rangle)\\
&- \varphi\ (\mathbf{lfst}\ d) - \textstyle\sum_{i=1}^n \varphi\ d_i + \sum_{i=1}^n \varphi\ env_i'\\
&+ \mathrm{cost}(\mathbf{run}\ (\mathrm{snd}\ \mathcal{D}_\Gamma[t_2]\ d)\ env;\\
&\qquad x_1 = x_1, \ldots, x_n = x_n, d = \mathbf{lsnd}\ d, env = env')\\
&- \varphi\ (\mathbf{lsnd}\ d) - \textstyle\sum_{i=1}^n \varphi\ env_i' + \sum_{i=1}^n \varphi\ env_i''\\
&\leq c \cdot \mathrm{cost}(t_1; x_1 = x_1, \ldots, x_n = x_n) + c \cdot \mathrm{cost}(t_2; x_1 = x_1, \ldots, x_n = x_n)\\
&\qquad + 2c' + 2c_{\mathrm{run}} \cdot n
\end{aligned} \tag{6.11}$$

Subtract Eq. (6.11) from Eq. (6.10):

$$\begin{aligned}
&O(1) - c_{\mathrm{run}} \cdot n - \varphi\ d + \varphi\ (\mathbf{lfst}\ d) + \varphi\ (\mathbf{lsnd}\ d) - \textstyle\sum_{i=1}^n \varphi\ env_i'' + \sum_{i=1}^n \varphi\ res_i\\
&\quad \leq c \cdot 1 - c' - c_{\mathrm{run}} \cdot n
\end{aligned}$$

where we simplified the right-hand side using our cost model, which gives that $\mathrm{cost}(\langle t_1, t_2\rangle; \Gamma) = 1 + \mathrm{cost}(t_1; \Gamma) + \mathrm{cost}(t_2; \Gamma)$. We can further simplify by observing that $env'' = res$ and by cancelling the two occurrences of $c_{\mathrm{run}} \cdot n$.

Then, to handle the $\varphi\ d$ terms, we need to consider sparsity: we need to analyse the cases where $d = \mathrm{inl}\ \langle\rangle$ and where $d = \mathrm{inr}\ \langle d_1', d_2'\rangle$. In the first case, we have $\mathbf{lfst}\ d = \underline{0}$ and $\mathbf{lsnd}\ d = \underline{0}$, hence $\varphi\ d = \varphi\ (\mathbf{lfst}\ d) = \varphi\ (\mathbf{lsnd}\ d) = 1$, thus $-\varphi\ d + \varphi\ (\mathbf{lfst}\ d) + \varphi\ (\mathbf{lsnd}\ d) = 1 = O(1)$. In the second case, $\varphi\ d = 1 + \varphi\ (\mathbf{lfst}\ d) + \varphi\ (\mathbf{lfst}\ d)$, so the same expression evaluates to -1, which is also $O(1)$. Hence we can merge the three φ-terms into the $O(1)$ that is already there, yielding:

$$O(1) \leq c - c'$$

which is clearly true for sufficiently large c.

Again, note what happened to the potential. The potential in the incoming cotangent, d, was mostly immaterial: it contributed $+1$ or -1 depending on how sparsely it was represented, but did not do anything significant. This is expected, since in $\mathcal{D}_\Gamma[\langle t_1, t_2\rangle]$ we do not build or consume any non-trivial fragments of cotangent values. As for the potential in the environment cotangent accumulator: the outgoing potential of snd $\mathcal{D}_\Gamma[t_1]$, equal to $\sum_{i=1}^n \varphi\ env'_i$, cancels precisely against the incoming potential (via the environment cotangent) of snd $\mathcal{D}_\Gamma[t_2]$, which is again expected because the environment cotangent itself is passed as-is from t_1 to t_2.

In general, this is what always happens in the proof: as long as we do not do anything material to cotangents, they at most consume a bounded number of evaluation steps in stored potential, and as long as we do not modify the environment cotangent ourselves, all the corresponding φ terms cancel. The only case where we do something material to all of these, and where the φ terms do not immediately cancel, is for variable references.

Variable references: amortisation. Taking $t = x_i : \sigma_i$ in the environment $x_1 : \sigma_1, \ldots, x_n : \sigma_n$ and inlining into Eq. (6.7), we get to prove:

$$\begin{aligned}
&\exists c, c' > 0.\ \forall v : \sigma_i.\ \forall d : \mathcal{D}[\sigma_i]_2.\ \forall d_1 : \mathcal{D}[\sigma_1]_2, \ldots, d_n : \mathcal{D}[\sigma_n]_2. \\
&\quad O(1) + \mathrm{cost}(\mathbf{run}\ (\mathbf{one}_{x_i:\mathcal{D}[\sigma_i]_2 \in \mathcal{D}[\Gamma]_2} d)\ env_0; \\
&\qquad\qquad d = d, env_0 = \langle\langle\langle\rangle, d_1\rangle, \ldots\rangle, d_n\rangle) \\
&\qquad - \varphi\ d - \textstyle\sum_{j=1}^n \varphi\ d_j + \sum_{j=1}^n \varphi\ res_j \\
&\quad \le c \cdot \mathrm{cost}(x_i; x_i = v) + c' + c_{\mathrm{run}} \cdot n
\end{aligned} \tag{6.12}$$

Here we use v for the value of the variable x_i in the current evaluation environment. In our cost model, $\mathrm{cost}(x_i; \Gamma) = 1$ for a variable x_i, and furthermore we know that that for all $j \neq i$, we have $res_j = d_j$. For res_i, we know from the semantics of **one** that $res_i = d + d_i$. The complexity property of **one** specialises to the following:

$$\begin{aligned}
&\mathrm{cost}(\mathbf{run}\ (\mathbf{one}_{x_i:\mathcal{D}[\sigma_i]_2 \in \mathcal{D}[\Gamma]_2}\ d)\ env_0; d = d, env_0 = \langle\langle\langle\rangle, d_1\rangle, \ldots\rangle, d_n\rangle) \\
&\quad = O(1) + \mathrm{cost}(d + d_i; d = d, d_i = d_i)
\end{aligned}$$

Thus Eq. (6.12) simplifies to:

$$\begin{aligned}
&\exists c, c' > 0.\ \forall d : \mathcal{D}[\sigma_i]_2.\ \forall d_i : \mathcal{D}[\sigma_i]_2. \\
&\quad O(1) + \mathrm{cost}(d + d_i; d = d, d_i = d_i) - \varphi\ d - \varphi\ d_i + \varphi\ (d + d_i) \\
&\qquad \le c + c' + c_{\mathrm{run}} \cdot n
\end{aligned} \tag{6.13}$$

Now we use the central amortisation property of $(+)$ (Eq. (6.5) in Section 6.3.3), which implies that:

$$\mathrm{cost}(d + d_i; d = d, d_i = d_i) - \varphi\ d - \varphi\ d_i + \varphi\ (d + d_i) \le 0 \tag{6.14}$$

Subtracting Eq. (6.14) from Eq. (6.13) gives us that it is enough to show that $O(1) \leq c + c' + c_{\text{run}} \cdot n$, which is immediate for sufficiently large c.

Unlike before, we have actually used potential here: we received potential for d and d_i and needed to return potential for their sum. Because the sum will contain less potential than the inputs to (+), we can use the excess potential to pay for the execution of (+) itself, without needing to count more than a bounded number of evaluation steps here for the transform of a variable reference.

The other cases. The other cases in the proof are mostly analogous to the cases discussed above. For **let** $x = t_1$ **in** t_2, we end up needing the lemma that the CHAD primal of a term is equal to the result of the original term (i.e. that fst $\mathcal{D}_\Gamma[t]$ returns the same result as t when run in the same environment); this is required because we need to relate the cost of evaluating t_2 to that of evaluating **run** (snd $\mathcal{D}_\Gamma[t_2]$ d) env_0, and these two evaluations happen in the same environment only if fst $\mathcal{D}_\Gamma[t_1]$ and t_1 return the same result.

6.5.2 Agda formalisation

We have formalised the above complexity proof in Agda (≥2.6.3, also compiles with `--safe --without-K`). Agda [Norell 2007] is a dependently-typed functional language and proof assistant, and one of the standard proof assistants in the domain of programming languages. While not typically used for proofs with integer reasoning, it admits a very natural encoding of the problem statement. Our full development can be found online[17] and archived at [Smeding and Vákár 2023b]; the *statements* of the theorems, and the definitions required to write those statements, are included in Section 6.10.4.

In the development, the source and target language are encoded as a fully well-typed well-scoped (De Bruijn) inductive data type in the standard fashion; the cost model is encoded in the evaluator (`eval`), which evaluates an expression of type τ to a (meta-language, i.e. Agda) value of type $[\![\tau]\!] \times \mathbb{Z}$. The integer contains the number of 'steps' taken in evaluating the expression: our cost model.

In a way, the Agda proof is somewhat more generic than the sketch above, because it defines the methods of **EVM**[18] together with properties about their semantics and complexity in an `abstract` block. This means that the rest of the proof cannot use the *implementation* of the methods and the monad type itself, but only their *types* (and the properties of those methods that we provide in the block). Because all three of the monad implementations that we outlined in Section 6.4 satisfy those properties, we know that our Agda proof works for all

[17] `https://github.com/tomsmeding/efficient-chad-agda`

[18] The concrete implementation there called `LACM` for "Local ACcumulation Monad".

three, regardless of exactly which concrete monad implementation we choose for the Agda formalisation.[19]

The formalisation produced for this work was extended by a student to additionally prove semantical correctness of first-order CHAD by showing it equivalent to a postulated encoding of basic differential geometry. [Meijs 2025] Their work additionally includes a set-up for also proving correct the extension to higher-order programs of Section 6.8 using a variant of closure conversion (Section 6.10.2) by means of a logical relation.

6.6 Arrays

Adding arrays to our language, which is so far simply first-order, is not difficult, if somewhat tedious. We will show how the elements of a complexity proof for array operations are analogous to the cases already discussed in Section 6.5, but we leave a full proof to future work.

An array is a product type, hence we should take inspiration from the handling of binary products ($\sigma \times \tau$), where we introduced sparsity in order to efficiently represent zeros $\underline{0}_{\mathcal{D}[\sigma\times\tau]_2}$ and one-hot cotangents $\langle x, \underline{0}_\tau\rangle$ and $\langle \underline{0}_\sigma, x\rangle$. Thus our choice of $\mathcal{D}[\text{Array } \tau]_2$ should allow efficient zeros and one-hots as well.

A one-hot array cotangent is a pair of an index (of type $\mathbb{Z}$) and a cotangent for that cell (of type $\mathcal{D}[\tau]_2$), hence a sufficient choice seems to be to let $\mathcal{D}[\text{Array } \tau]_2$ be some collection Bag $(\mathbb{Z} \times \mathcal{D}[\tau]_2)$ of pairs that we will convert to an array of pairs using collect : Bag $\tau \to$ Array τ once we are done constructing it. For efficient differentiation of discarding, indexing and sharing, this bag should furthermore support constant-time creation of empty and singleton collections, as well as constant-time combination of two collections. It turns out to be sufficient to simply defunctionalise the operations that we want to be efficient and use the following type definition (i.e. make them constant-time by construction):[20]

$$\textbf{data}\ \text{Bag}\ \tau = \text{BEmpty} \mid \text{BOne}\ \tau \mid \text{BPlus}\ (\text{Bag}\ \tau)\ (\text{Bag}\ \tau)$$

Observe that 'collect'ing such a Bag τ costs at most as much time as was spent constructing it. Hence, if we use the Bag τ affinely, which we do, the cost of 'collect'ing it can be amortised against its creation. So in terms of asymptotic complexity, we cannot do better than this, absent parallelism.

This leads to the definitions $\mathcal{D}[\text{Array } \tau]_1 = \text{Array } \mathcal{D}[\tau]_1$ and $\mathcal{D}[\text{Array } \tau]_2 = \text{Bag } (\mathbb{Z} \times \mathcal{D}[\tau]_2)$. We still have $\mathcal{D}[\tau]_1 = \tau$ for all types.

[19] The actual Agda implementation is a state monad with cons-list state, but with costs as if it was a constant-time version.

[20] The fact that Bag is a free monoid is unsurprising (because among the operations we defunctionalised are zero and plus), but also not very fundamental: in Section 6.7.1 we will add more constructors to Bag to avoid pessimising other operations.

Operations. We discuss three array operations here: elementwise construction, indexing and associative reduction. Their typing rules are as follows:[21]

$$\frac{\Gamma \vdash s : \mathbb{Z} \quad \Gamma, i : \mathbb{Z} \vdash t : \tau}{\Gamma \vdash \text{build } s\ (i.\ t) : \text{Array } \tau} \qquad \frac{\Gamma \vdash s : \text{Array } \tau \quad \Gamma \vdash t : \mathbb{Z}}{\Gamma \vdash s\ !\ t : \tau}$$

$$\frac{\Gamma, x : \tau \times \tau \vdash s : \tau \quad \Gamma \vdash t : \text{Array } \tau}{\Gamma \vdash \text{fold } (x.\ s)\ t : \tau}$$

The informal semantics are as follows: build $n\ (i.\ t) = [t[0/i], \ldots, t[n-1/i]]$, $[x_1, \ldots, x_n]\ !\ i = x_i$, and fold $(x.\ \text{fst}\ x \star \text{snd}\ x)\ [x_1, \ldots, x_n] = x_1 \star \cdots \star x_n$. 'fold' requires its array argument to be of non-zero length, and performs a reduction in unspecified order, assuming that t is associative.[22]

Parallel array languages typically have other operations as well, including special cases of 'build' such as gather, transpose, stencils/convolutions and more (which can be implemented more efficiently than the general case), but also independent operations such as various scans as well as scatter (forward array permutation). In practice one will need a specialised derivative for each of these, the former for efficiency and the latter for expressivity, but here we restrict ourselves to the given three, which are together already powerful enough to express most machine learning models.

Derivatives. We show the derivatives of the three operations in Fig. 6.8. The simplest of the three, $\mathcal{D}_\Gamma[s\ !\ t]$, should not be surprising given the choice of $\mathcal{D}[\text{Array } \tau]_2$: it behaves similarly to the derivative of a tuple projection (fst and snd), and its complexity is clearly sound for the same reasons. For build, we use three additional array operations with the following types:

$$\text{unzip} : \text{Array } (\sigma \times \tau) \to (\text{Array } \sigma) \times (\text{Array } \tau)$$
$$\text{scatter} : \text{Monoid } \tau \Rightarrow \text{Array } \tau \to \text{Array } (\mathbb{Z} \times \tau) \to \text{Array } \tau$$
$$\frac{\Gamma, x : \sigma, y : \tau \vdash r : \rho \qquad s : \text{Array } \sigma \qquad t : \text{Array } \tau}{\Gamma \vdash \text{zipWith } (x\ y.\ r)\ s\ t : \text{Array } \rho}$$

where 'scatter' "adds" the values in its second argument to the indicated positions in the first argument using the "add" operation from its monoid structure. (The 'Monoid $\tau \Rightarrow$' notation indicates a constraint on τ, using Haskell syntax.) 'unzip' and 'zipWith' can be defined in terms of 'build' and indexing; 'scatter' is a new primitive running in time linear in the size of its inputs in the sequential case.

[21] We elide λ in the syntax to emphasise that we do not (yet) allow unrestricted lambda abstraction.

[22] This is a typical operation in parallel array languages, such as `fold` in Accelerate and `reduce` in Futhark. The requirement that the array be non-empty is typically lifted by adding an additional initial value for the reduction, but that would make this section more verbose without adding interesting new problems.

$$
\begin{aligned}
&\mathcal{D}_\Gamma[\text{build } s\ (i.\ t : \tau)] = \\
&\quad \textbf{let } \langle n, \_\rangle = \mathcal{D}_\Gamma[s] \\
&\qquad a = \text{build } n\ (i.\ \mathcal{D}_{\Gamma, i:\mathbb{Z}}[t]) \\
&\qquad \langle a_1, a_2\rangle = \text{unzip } a \\
&\quad \textbf{in } \langle a_1, \lambda d.\ \textbf{let } \mathit{pairs} = \text{collect } d \\
&\qquad\qquad\qquad d_2 = \text{scatter } (\text{build } n\ (i.\ \underline{0}_{\mathcal{D}[\tau]_2}))\ \mathit{pairs} \\
&\qquad\qquad\qquad d_3 = \text{zipWith } (f\ d'.\ \textbf{scope}_{\mathcal{D}[\Gamma]_2, \underline{\mathbf{1}}}\ (f\ d') \gg \textbf{return } \langle\rangle)\ a_2\ d_2 \\
&\qquad\qquad \textbf{in } \text{sequence } d_3 \gg \textbf{return } \langle\rangle\rangle \\
&\mathcal{D}_\Gamma[s\ !\ t] = \textbf{let } \langle x_1, x_2\rangle = \mathcal{D}_\Gamma[s]; \langle i, \_\rangle = \mathcal{D}_\Gamma[t] \\
&\qquad\qquad \textbf{in } \langle x_1\ !\ i, \lambda d.\ x_2\ (\text{BOne } \langle i, d\rangle)\rangle \\
&\mathcal{D}_\Gamma[\text{fold } (p.\ s)\ (t : \text{Array } \tau)] = \\
&\quad \textbf{let } \langle t_1, t_2\rangle = \mathcal{D}_\Gamma[t] \\
&\qquad \mathit{tree} = \text{fold } (p'.\ \textbf{let } p = \langle \text{getA } (\text{fst } p'), \text{getA } (\text{snd } p')\rangle \\
&\qquad\qquad\qquad\qquad \langle y, f\rangle = \mathcal{D}_{\Gamma, p:\tau\times\tau}[s] \\
&\qquad\qquad\qquad\quad \textbf{in } \text{Node } (\text{fst } p')\ y\ f\ (\text{snd } p')) \\
&\qquad\qquad\qquad (\text{map } (x.\ \text{Leaf } x)\ t_1) \\
&\quad \textbf{in } \langle \text{getA } \mathit{tree} \\
&\qquad ,\lambda d.\ \textbf{do } \mathit{lf} \leftarrow \text{unTree } (\lambda d'\ f. \\
&\qquad\qquad\qquad\qquad\qquad\qquad \textbf{do } \langle\langle\rangle, \langle d_1, d_2\rangle\rangle \leftarrow \textbf{scope}_{\mathcal{D}[\Gamma]_2, \mathcal{D}[\tau\times\tau]_2}\ (f\ d') \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{return } \langle d_1, d_2\rangle) \\
&\qquad\qquad\qquad\qquad\qquad\quad d\ \mathit{tree} \\
&\qquad\qquad\quad t_2\ (\text{fromList } (\mathit{lf}\ [\,]))\rangle
\end{aligned}
$$

Figure 6.8: The derivative of the build, array indexing and fold operators.

$$
\begin{aligned}
&\textbf{data } \text{Tree } a\ f = \text{Node } (\text{Tree } a\ f)\ a\ f\ (\text{Tree } a\ f) \mid \text{Leaf } a \\
&\text{getA} : \text{Tree } a\ f \to a \\
&\text{getA } (\text{Node } \_\ x\ \_\ \_) = x \\
&\text{getA } (\text{Leaf } x) = x \\
&\text{unTree} : \text{Monad } m \Rightarrow (d \to f \to m\ (d \times d)) \to d \to \text{Tree } a\ f \\
&\qquad\qquad\qquad\qquad \to m\ (\text{List } d \to \text{List } d) \\
&\text{unTree } g\ d\ (\text{Node } t_1\ \_\ f\ t_2) = \textbf{do } \langle d_1, d_2\rangle \leftarrow g\ d\ f \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad rs_1 \leftarrow \text{unTree } g\ d_1\ t_1 \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad rs_2 \leftarrow \text{unTree } g\ d_2\ t_2 \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{return } (\lambda l.\ rs_1\ (rs_2\ l)) \\
&\text{unTree } g\ d\ (\text{Leaf } \_) = \textbf{return } (\lambda l.\ d :: l) \qquad -- (::) \textit{ is list cons}
\end{aligned}
$$

Figure 6.9: The definitions of Tree and unTree used in $\mathcal{D}_\Gamma$[fold]. List α are cons-lists of α; (::) is their cons operator.

Finally we also use collect : Bag $\tau \to$ Array τ and the monadic sequence operation (sequence : Array ($\mathbf{EVM}\ \Gamma\ \tau$) $\to \mathbf{EVM}\ \Gamma$ (Array τ)).[23]

In the primal of $\mathcal{D}_\Gamma[\text{build}\ s\ (i.\ t)]$, we simply build an array of the primal results of t for each i, return the array of first components as the primal result, and retain the array of backpropagators for use in the backpropagator of 'build'.[24] Then, when the backpropagator is called, we receive a sparse array cotangent in the form of a Bag of pairs. After eliminating the Bag structure, we construct the full cotangent using 'scatter' (in $O(n)$), and then we run each of the element backpropagators on the corresponding cotangent from d_2. Finally, 'sequence' runs all the resulting monad actions. The appearance of 'scatter' (forward array permutation) is unsurprising, because 'build' is the quintessential gathering (backward array permutation) operation, and reverse differentiation dualises data flow.

As for the complexity for 'build': all array operations in $\mathcal{D}_\Gamma[\text{build}\ s\ (i.\ t)]$ operate on arrays of the same length and can run in linear time in the sequential setting — in addition to the expected invocations of the backpropagators f. The derivative functions resulting from the execution of $\mathcal{D}_\Gamma[s]$ and $\mathcal{D}_{\Gamma,i:\mathbb{Z}}[t]$ are executed at most once, and cotangent values are treated affinely as required. Hence, the complexity proof should extend analogously to the cases shown in Section 6.5.1.

The derivative of 'fold' in Fig. 6.8 is a bit more involved.[25] The approach taken here is to record (in a 'Tree' — see Fig. 6.9) the reduction tree taken in the primal pass by the 'fold' combinator, and to unfold over that same reduction tree, but now from the root instead of from the leaves, in the reverse pass (with 'unTree')[26]. In practice, one would implement 'unTree' as a primitive operation together with the 'Tree' data type, and hide this complexity from users. (See also Section 6.7.) Finally we also use a new array primitive: fromList : List $\tau \to$ Array τ, clearly also linear-time in the length of the list. The complexity for fold is sound for the same reasons as for 'build' above: its direct work is within bounds, and it calls backpropagators of its subterms at most (here precisely) once.

[23] Sequencing is what seems to preclude a parallel implementation here, but see Section 6.7.

[24] Note that snd $\mathcal{D}_\Gamma[s]$ is thrown away, because being a linear function with $\underline{\mathbf{1}}$ as domain, it cannot compute anything useful. Said differently, s being of discrete type ($\mathbb{Z}$), it cannot continuously depend on anything.

[25] An alternative is given by Paszke et al. [2021b]; see Section 6.9.

[26] The 'List $d \to$ List d' type is a Cayley-transformed list / "difference list" for constant-time concatenation [Hughes 1986].

6.7 Parallelism and practical efficiency

Automatic differentiation is typically applied to programs with inherent parallelism, so it would be a shame if the derivative program was forced to be sequential. So far, the prime inhibitor to parallelisation of the derivative program seems to be the monad **EVM**. Of course, we cannot run the left-hand and right-hand side of a bind operation ($\gg=$) in parallel, but in our code transformation (see Figs. 6.6 and 6.8) such binding is only used when there was an actual dependency in the source program already, for example due to a let-binding. For independent source expressions, the corresponding monad actions are independently sequenced using ($\gg$) and 'sequence'.

As we observed before in Section 6.3.2, this allows parallelism in the derivative program. Our monad being a (local) accumulation monad, all updates to the individual cells *add* a new contribution to the value already in that cell; since addition is commutative and associative, it does not matter in which order we add these contributions: the inevitable reordering resulting from concurrent updates is fine. We just need to ensure that the individual contributions do not get corrupted by concurrent access to the same mutable cells; for this, locks or atomic updates suffice.[27]

Reimplementing the API of **EVM** in this way, it becomes safe to execute the monadic actions sequenced with ($\gg$) and 'sequence' in parallel, resulting in parallelism corresponding to independent expressions in the source program. In an actual implementation it would be prudent to have both sequential and parallel versions of ($\gg$) and 'sequence', because for some uses, parallelisation will cost more in overhead than it gains in useful parallelism.

This covers all operations expressed with the mentioned parallelisable combinators; what is left is the unTree function from Fig. 6.9, which we need to execute in such a way that the parallelism of the original 'fold' reduction is reflected in its derivative. Fortunately, it suffices to execute the two recursive calls to unTree in parallel: this is possible because they are independent (which one could formalise by putting them in a 2-element array to 'sequence' or by using applicative functor operators). The resulting task parallelism mirrors the reduction tree structure of the original 'fold', and can thus also express the parallelism of the original reduction.

Complexity. Unlike the sequential case, it is unclear what a proper complexity criterion should be for the parallel case. It is not hard to see that the total amount of *work* performed even in the parallel derivative is proportional to that of the

[27] Locks are tricky here: they must not be too fine-grained (e.g. around individual cotangent scalars) nor too coarse-grained (e.g. around derivatives of large context variables). Using atomic updates to individual scalars/pointers is more straightforward.

input program, but requiring a constant factor slowdown over the source program in overall runtime (the *span* of the program) is impossible: parallel replication (build n $(i.\ x)$) seemingly has, in a naive cost model, constant runtime given enough parallel execution units (i.e. constant span), whereas its derivative, which must perform a parallel reduction (summing all entries in the incoming cotangent value), surely has span at least logarithmic in n. Note that this n need not be a visible, or even easily computable, property of the source program if it is a computed value, making it hard to even formulate the optimal complexity criterion for reverse AD on parallel array programs. For this reason, we leave a formal complexity analysis of the parallel case to future work.

6.7.1 Constant factors and execution on vector machines

The approach described above will work acceptably on multicore CPU platforms with relatively low core counts, once some care has been taken to avoid excessive parallelism overhead by switching to sequential execution for subexpressions that are already executed in a sufficiently parallel manner. However, to work on GPU/TPU platforms or similar wide-vector machines, as well as to gain more performance on MIMD architectures, it will be necessary to:

1. find alternate implementations of the tree-like structures: Bag (Section 6.6) and Tree (Fig. 6.9);

2. analyse and optimise the output derivative program, recognising places where our base transformation was too general and a special-case approach would be more efficient.

'Tree' on vector machines. In $\mathcal{D}_\Gamma$[fold], 'Tree' is used to record the reduction tree in the primal pass so that we can replay it in reverse order in the reverse pass. In practice on wide vector machines such as GPUs, the reduction tree structure of a fold is statically determined to a certain extent. For example, for the (still competitive) approach described by Merrill and Garland [2016] (see their Fig. 5), despite the fact that the block aggregates are combined in some nondeterministic order, the tree of intermediate values corresponding to the more classical chained-scan approach (their Fig. 4) is still computed and stored. Keeping these stored intermediate values around until the reverse pass allows assuming the chained-scan reduction order in the reverse pass, meaning that we only need to store the block size (an integer), the block aggregates and the block-local sequential aggregates, where our Fig. 6.8 stored the full 'Tree'. In effect, we thus specialise 'Tree' to the practical (strongly reduced) space of possible reduction orders in the actual implementation, and choose a more compact — and in this case non-recursive! — representation for 'Tree' that describes just this

smaller space. Doing so allows us to convert the task parallelism in unTree to data parallelism (for suitable combination functions), mirroring the data-parallel reduction in the primal pass.

'Bag' on vector machines. First note that we may only *add* constructors to Bag, not subtract, as we certainly need the current ones (zero, plus, and singleton) for general source programs that use array indexing. But sometimes we can, and need to, do better. For example, when differentiating the following program, which first computes some array a and then multiplies a by 2 pointwise:[28]

$$\ldots \vdash \textbf{let } a = \text{build } (\ldots)\ (i.\ \ldots) \textbf{ in } \text{build } (\text{length } a)\ (i.\ (a\,!\,i) \cdot 2) : \text{Array } \tau \quad (6.15)$$

the derivative program will create (in snd $\mathcal{D}_\Gamma[a\,!\,i]$, which is called 'length a' times) many BOne values, add those together into a large tree (in 'sequence' in the $\mathcal{D}_\Gamma[\text{build}]$ for the result, thus — in a parallel context — of nondeterministic associativity), serialise that tree to an array (with 'collect' in the $\mathcal{D}_\Gamma[\text{build}]$ of a), and finally perform a 'scatter' to construct the gradient of a. However, clearly a more efficient derivative is to simply multiply the result cotangent by 2 pointwise, and while what we generate is indeed "only" a constant factor off in a sequential setting, this constant factor is in fact very large, and furthermore it parallelises poorly.

This program exhibits a pattern known as a *gather* operation,[29] which is the program shape 'build n $(i.\ s\,!\,t)$' where i does not occur freely in s. In this case, aside from the operations that we already made constant-time by construction by making them constructors of Bag directly (zero, plus, and singleton), we also want to be able to insert a full array of cotangents.[30] Thus we can improve the situation with a principled change to our data structure, adding a constructor (BArray (Array τ)) to the Bag data type. To then productively use this constructor in differentiating the sample program in Eq. (6.15), the implementation should either recognise the gather shape of the source program pre-differentiation and rewrite it to use some gather-style primitive, or should recognise the (inefficient) pattern resulting from naively differentiating a gather-like build and optimise that to the special-purpose form using BArray.

[28] Assuming the addition of length : Array $\tau \to \mathbb{Z}$ in the source language; because its return type is discrete, its derivative is trivial: $\mathcal{D}_\Gamma[\text{length } t] = \langle \text{length } (\text{fst } \mathcal{D}_\Gamma[t]), \lambda\_.\ \textbf{return } \langle\rangle \rangle$.

[29] The derivative of 'gather' is 'scatter', already used above. Using a single scatter, even if no more efficient form is found based on the particular index mapping used in the program, will be much faster than creating large numbers of BOne values and having to combine those in a log-depth tree with many uses of BPlus.

[30] This constructor will be constant-time, but its "virtual cost" will be inflated to the size of the array, so that there is sufficient potential in the Bag for 'collect' to amortise against later. This linear cost is acceptable because BArray will replace other linear-time operations.

As with all compiler optimisations, however, such tricks cannot cover all possible programs, but the common cases can be dealt with in this manner. In the next chapter on Fast CHAD, and specifically in Section 7.2, we improve on this situation in a radical way: we drop the use of 'Bag' entirely and revert to a more naive $\mathcal{D}[\text{Array } \tau]_2 = \mathbf{1} \sqcup \text{Array } \mathcal{D}[\tau]_2$. We avoid reintroducing one-hot arrays for indexing by restricting the input language and extending **one** to allow mutation of individual elements inside arrays in the environment cotangent collector.

6.8 Function types

As a final contribution in this chapter, we extend the complexity-efficient algorithm to the full language of Chapter 5 by adding function types. As before, we first do so naively, then we identify the complexity problems and subsequently we solve them.

Naive CHAD for function types. As a reminder and to update them to the presentation in this chapter, we present the naive CHAD rules for differentiating function types again. We write [] for an empty list, ++ for concatenation and 'foldl' for the usual sequential left fold: foldl $(\star)$ x_0 $[x_1, \ldots, x_n] = ((x_0 \star x_1) \star \cdots) \star x_n$.

$$\mathcal{D}[\sigma \to \tau]_1 = \mathcal{D}[\sigma]_1 \to (\mathcal{D}[\tau]_1 \times (\mathcal{D}[\tau]_2 \to \mathcal{D}[\sigma]_2))$$
$$\mathcal{D}[\sigma \to \tau]_2 = \text{List } (\mathcal{D}[\sigma]_1 \times \mathcal{D}[\tau]_2)$$
$$\underline{0}_{\text{List } (\mathcal{D}[\sigma]_1 \times \mathcal{D}[\tau]_2)} = [] \qquad s +_{\text{List } (\mathcal{D}[\sigma]_1 \times \mathcal{D}[\tau]_2)} t = s +\!\!+ t$$

$$\begin{aligned}
&\mathcal{D}_\Gamma[\lambda x : \tau.\ t] = \\
&\quad \textbf{let } f = \lambda x.\ \mathcal{D}_{\Gamma, x:\tau}[t] \\
&\quad \textbf{in } \langle \lambda(x : \mathcal{D}[\tau]_1).\ \textbf{let } \langle y, y' \rangle = f\ x \\
&\qquad\qquad \textbf{in } \langle y, \lambda d.\ \text{snd } (\textbf{split}_{\mathcal{D}[\Gamma]_2, \mathcal{D}[\tau]_2}\ (y'\ d)) \rangle, \\
&\qquad , \lambda d.\ \text{foldl } (\lambda acc.\ \lambda \langle x, d' \rangle.\ acc + \text{fst } (\textbf{split}_{\mathcal{D}[\Gamma]_2, \mathcal{D}[\tau]_2}\ (\text{snd } (f\ x)\ d'))) \\
&\qquad\qquad \underline{0}_{\mathcal{D}[\Gamma]_2}\ d \rangle \\
&\mathcal{D}_\Gamma[s\ t] = \textbf{let } \langle f, f' \rangle = \mathcal{D}_\Gamma[s] \\
&\qquad\qquad\quad \langle x, x' \rangle = \mathcal{D}_\Gamma[t] \\
&\qquad\qquad\quad \langle y, y' \rangle = f\ x \\
&\qquad\qquad \textbf{in } \langle y, \lambda d.\ f'\ [\langle x, d \rangle] + x'\ (y'\ d) \rangle
\end{aligned}$$

Note the explicit use of **split**, which was unnecessary in Fig. 5.4.

The key observation here, as already made in Chapter 5, is that in a function application, the incoming cotangent must be propagated backwards through the λ-abstraction being called, *both to the function argument and to the captured context variables of the closure.* CHAD naturally separates these two parts of the derivative and handles the former with $\mathcal{D}[\sigma \to \tau]_1$ and the latter with

$\mathcal{D}[\sigma \to \tau]_2 \to \mathbf{EVM}\ \mathcal{D}[\Gamma]_2\ \mathbf{1}$, which is the type of 'snd $\mathcal{D}_\Gamma[t]$' if t is of type $\sigma \to \tau$. The list $\mathcal{D}[\sigma \to \tau]_2$ is a log of all invocations of the function, containing for each invocation its input primal and its output cotangent.

Identifying the complexity issues. It is precisely this separation of the derivative that leads to real[31] complexity problems. In particular, because the derivative of a function value is split in two parts, it is impossible (from the output of $\mathcal{D}_\Gamma[\lambda x.\ t]$) to get the full gradient of a function, i.e. with respect to both its argument and its context, in one pass through $\mathcal{D}_{\Gamma,x:\tau}[t]$. And for function application, which is the only eliminator of functions in the source language, snd $\mathcal{D}_\Gamma[s\ t]$ does indeed need the full gradient of the function that was called (i.e. t). It must therefore resort to using both halves of the function's derivative separately, meaning that we end up differentiating through a function *twice* each time it is called. If that function contains other function applications inside its body, the derivatives of the functions called there are evaluated four times, etc.

This behaviour can be exploited to violate our complexity criterion Eq. (6.2). Indeed, consider the programs t_n for each n (which are nested identity applications, and thus semantically just the identity):[32]

$$t_n \quad \coloneqq \quad x_n : \mathbb{R} \vdash (\lambda x_{n-1}.\ (\cdots \lambda x_2.\ (\lambda x_1.\ x_1)\ x_2 \cdots)\ x_{n-1})\ x_n : \mathbb{R}$$

Then t_n runs in $O(n)$ time. However, their transposed derivatives snd $\mathcal{D}_{x_n:\mathbb{R}}[t_n]$ using the CHAD formulas above take $O(2^n)$ time to execute: because they contain n nested pairs of an application of an abstraction, the backpropagator of each will execute the backpropagator of its body twice.

Solving complexity issues through defunctionalisation. Clearly, defunctionalisation [Reynolds 1998] translates away function types into a language that we can already differentiate efficiently, by implementing function types $\sigma \to \tau$ as a sum type of tuples $\rho^\ell_1 \times \cdots \times \rho^\ell_{n_\ell}$ for each syntactic lambda-abstraction ℓ of type $\sigma \to \tau$ in the program, writing $\rho^\ell_1, \ldots, \rho^\ell_{n_\ell}$ for the list of types of ℓ's captured context variables. (This list is a subset of the types in ℓ's environment.) As such, we can simply defunctionalise (a well-known strategy for compiling code with function types) before applying CHAD and then call it a day. But *why* exactly does this solve the problem of inefficient function types? The key observation is to decompose defunctionalisation into the composition of:

[31] Technically the linear-time cost of $+\!\!+$ is also a problem, but that can be resolved by Cayley-transforming or using Bag.

[32] It is unnecessary for the function being applied to be a *literal* lambda expression — defining the lambda somewhere and calling it elsewhere calls the same backpropagators in the end. The example is just written this way for conciseness.

- the local program transformation of (typed) closure conversion [Minamide et al. 1996]: we convert every function into a closure, which is a pair of (a subset of) its environment and a function that does not capture any context variables (a "closed" function);

- the global program transformation of "deexistentialisation" that replaces an existential type with the finite sum type of all of its instantiations found in the whole program. A global program analysis is needed here to be able to use a finite rather than an infinite sum type: we need to analyse precisely which instances of the existential are actually used.

As we show in Section 6.10.2, it is the first part of the transformation that solves the efficiency problems of CHAD on function types: replacing all functions with closed functions. As a consequence, we avoid the need to propagate back any cotangents to captured context variables, removing the need for one half of a function's CHAD derivative; with only one half left, the duplication is gone, eliminating the complexity problem. In particular, it will now suffice to take $\mathcal{D}[\sigma \to \tau]_2 = \underline{\mathbf{1}}$, which simplifies the term-level derivatives accordingly. The resulting CHAD transformation remains local.

This idea of using closure conversion to speed up AD of higher-order functions is first used by Pearlmutter and Siskind [2008] (later distilled to its essence by Alvarez-Picallo et al. [2021]). More recently, in a short paper, Vytiniotis et al. [2019] suggested its use in the context of CHAD-like AD transformations. This section can be seen as an elaboration of the suggested idea of the latter paper.

6.9 Related work

This chapter shows how the basic reverse AD algorithm of CHAD can be made efficient. The basic CHAD technique for a first-order language with tuples in combinator form was originally introduced by Elliott [2018]. [Vákár 2021; Vákár and Smeding 2022; Nunes and Vákár 2023] show how it applies to a lambda calculus with various type formers, giving a correctness proof, but no complexity proof. Kerjean and Pédrot [2024] point out that the resulting code transformation closely resembles the Diller-Nahm variant of the Dialectica interpretation.

Mutability in functional AD tends to be used for accumulation: this occurs in Chapter 3 as well as Dex [Paszke et al. 2021a] and Futhark [Schenck et al. 2022]. (In [Radul et al. 2023], which describes the basic structure of Dex' AD algorithm (linearise-then-transpose), mutation is not yet necessary due to the simplicity of their input language.) Dex extends the method to a richer source language and needs to use mutability with an algebraic effect for (parallel) accumulation, similar

to the solution in this chapter; Futhark uses uniqueness types to implement the same idea. We instead use a monad to implement this effect.

Previous work in computer-formalised proofs about AD are, to the best of our knowledge, limited to [Chin Jen Sem 2020], which formalises the correctness proofs for the dual-numbers forward-mode AD transformation of [Shaikhha et al. 2019; Huot et al. 2020; Barthe et al. 2020; Vákár 2020; Huot et al. 2022; Lucatelli Nunes and Vákár 2024] in Coq, and work of de Vilhena and Pottier [2023], who give a Coq proof of the correctness of an effect handler-based variant of the reverse-mode AD techniques of [Wang et al. 2018, 2019] (which rely on non-functional control flow). Both papers focus on the correctness of AD, rather than its complexity.

In currently used industrial systems (such as TensorFlow [Abadi et al. 2016], PyTorch [Paszke et al. 2017] or JAX [Bradbury et al. 2018]), AD is typically performed on first-order (data-parallel) functional array processing languages. AD of second order functional array languages as a code transformation has been considered recently for Futhark by Schenck et al. [2022]. They allow some recompution and a resulting suboptimal complexity to achieve a simpler and more practically efficient algorithm, in the hope that such recomputation is rare in practice. By contrast, here we study how to avoid all recomputation. Paszke et al. [2021b] present a derivative for scans just in terms of standard second-order array combinators, but this version has the downside of being not quite complexity-efficient — it has a complexity blowup in the case of nesting fold in the combination function of fold. We avoid this blow-up with a custom primitive for the derivative (of fold, in our case, but we expect scans to work similarly).

The idea of using closure conversion to make AD of higher order functions efficient first appears buried in the details of VLAD/Stalin∇ [Pearlmutter and Siskind 2008; Siskind and Pearlmutter 2016]. The idea is again present in [Vytiniotis et al. 2019] (in the context of CHAD) and [Alvarez-Picallo et al. 2021] (for an AD algorithm using string diagram rewrites), without precisely demontrating its importance. We have made an effort to spell out and motivate the idea in this chapter, making clear how it arises as a natural solution to the complexity problems of higher-order CHAD.

Elsman et al. [2022] show that treating multi-linear operations as special cases can result in very nice derivatives using a generalised product-rule where the general approach produces unwieldy code. It would be interesting to see if CHAD can benefit from this insight in a principled way by making multi-linear operations more first-class, especially since their setting is similar (although expressed on a smaller language, lacking tuples, array indexing and first-class control flow).

Recently, Van den Berg et al. [2024] made clever use of type classes to present

various AD algorithms in a uniform way. Their considerations are orthogonal to our concerns in this thesis.

Shaikhha et al. [2023] discuss how to differentiate *source code* that uses sparse array operations efficiently. By contrast, we use a sparse array representation in the *generated derivative code* to achieve efficiency.

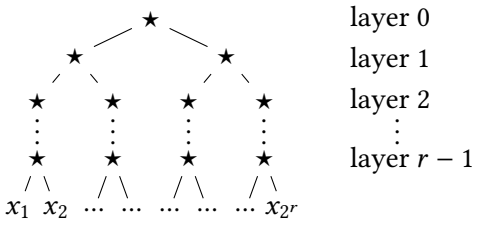
6.10 Appendices

6.10.1 Why *union* is not efficient

In Section 6.3.2 we changed the transformation so that the transformed code passes around a growing environment cotangent in state-passing style, instead of simply returning the local environment contributions upwards from each branch of the program. Not only does this provide the right program structure to later swap out the (log-time) functional persistent tree map for a mutable array in Section 6.4, but it is also necessary from a complexity perspective: simply keeping *union* is not efficient enough.

Map union[33] has runtime $O(m \log(\frac{n}{m} + 1))$, where m is the size of the smaller argument to *union* and n the size of the larger. For small $m > 0$ this simplifies to $O(\log n)$, and for $m \leq n$ in $O(n)$ (and in particular for $m = n$) it simplifies to $O(n)$.

Consider the term $x_1 : \mathbb{R}, \ldots, x_n : \mathbb{R} \vdash t_{\text{magic}} : \mathbb{R}$, defined as follows:



i.e. a complete binary tree combining $x_1, \ldots, x_{2^r}$ using a binary operation $\star$, where we set $r = \lfloor \log_2(n) \rfloor$ so that $2^r \leq n < 2^{r+1}$. All variables in the leaves are distinct. For $0 \leq i \leq r-1$, layer i has 2^i occurrences of $\star$, and hence in $\mathcal{D}_\Gamma[t_{\text{magic}}]$ we will get 2^i applications of *union* on maps of size 2^{r-1-i} on layer i. These unions are on arguments of equal size and thus run in time $O(2^{r-1-i})$.

The total computational cost of all these unions is:

$$\sum_{i=0}^{r-1} 2^i \cdot O(2^{r-1-i}) = O(r \cdot 2^r) = O(\log(n) \cdot n)$$

which is asymptotically larger than $O(n)$, the runtime cost of t_{magic}. (Note that the total number of $\star$ operations in t_{magic} itself is equal to $\sum_{i=0}^{r-1} 2^i = 2^r - 1 = O(n)$.) Hence, even with an optimal union implementation, CHAD does not yet attain the correct complexity for reverse AD and the state passing modification in Section 6.3.2 is necessary.

[33] As defined in the Haskell `containers` library, and proved optimal in [Brown and Tarjan 1979].

6.10.2 Function types: Closure conversion

The idea behind typed closure conversion [Minamide et al. 1996] is as follows. Introduce a type of closed functions $\sigma \xrightarrow{\square} \tau$ with the following typing rules:

$$\frac{x : \sigma \vdash t : \tau}{\Gamma \vdash \lambda x.\, t : \sigma \xrightarrow{\square} \tau} \qquad \frac{\Gamma \vdash s : \sigma \xrightarrow{\square} \tau \quad \Gamma \vdash t : \tau}{\Gamma \vdash s\, t : \tau}$$

The idea is that $\sigma \xrightarrow{\square} \tau$ only holds *closed* functions from σ to τ, i.e. ones that do not capture any context variables. Further, we write $\Sigma_{\alpha:\mathsf{Type}}\, \tau$ for a sum type indexed by the kind of types:[34]

$$\frac{\Gamma \vdash t : \tau[\sigma/\alpha]}{\Gamma \vdash \mathbf{pack}_\sigma\, t : \Sigma_{\alpha:\mathsf{Type}}\, \tau} \qquad \frac{\Gamma \vdash t : \Sigma_{\alpha:\mathsf{Type}}\, \tau \qquad \alpha : \mathsf{Type}, \Gamma, x : \tau \vdash s : \rho}{\Gamma \vdash \mathbf{case}\ t\ \mathbf{of}\ \mathbf{pack}_\alpha\, x \to s : \rho}$$

where α can occur freely in τ and s in the elimination rule. That is, we assume that we have an (impredicative) type universe Type with decidable equality in our type system. Such a universe can be implemented for our type system, for example, in Haskell by using GADTs.

Now, we transform function types using an existential type:

$$(\sigma \to \tau)^{CC} = \Sigma_{\alpha:\mathsf{Type}}\, \alpha \times ((\alpha \times \sigma^{CC}) \xrightarrow{\square} \tau^{CC})$$

and other types structurally recursively:

$$\mathbb{R}^{CC} = \mathbb{R} \quad \mathbf{1}^{CC} = \mathbf{1} \quad (\sigma \times \tau)^{CC} = \sigma^{CC} \times \tau^{CC} \quad (\sigma \sqcup \tau)^{CC} = \sigma^{CC} \sqcup \tau^{CC}$$

and, writing $FV(t)$ for the free variables occurring in the term t, we transform lambda abstraction and application:

$$\begin{aligned}(\lambda x.\, t)^{CC} &= \mathbf{pack}_{\mathsf{TypeOf}(\langle FV(t)\setminus\{x\}\rangle)^{CC}}\, \langle \langle FV(t) \setminus \{x\}\rangle, \lambda\langle\langle FV(t) \setminus \{x\}\rangle, x\rangle.\, t^{CC}\rangle \\ (s\, t)^{CC} &= \mathbf{case}\ s^{CC}\ \mathbf{of}\ \mathbf{pack}_\alpha\, z \to \mathbf{let}\ \langle cvars, f\rangle = z\ \mathbf{in}\ f\, \langle cvars, t^{CC}\rangle\end{aligned}$$

and other terms again structurally recursively:

$$\begin{aligned} x^{CC} &= x \qquad (\mathbf{let}\ x = s\ \mathbf{in}\ t)^{CC} = \mathbf{let}\ x = s^{CC}\ \mathbf{in}\ t^{CC} \\ \langle\rangle^{CC} &= \langle\rangle \qquad (\mathrm{fst}\ t)^{CC} = \mathrm{fst}\ t^{CC} \\ \langle s, t\rangle^{CC} &= \langle s^{CC}, t^{CC}\rangle \qquad (\mathrm{snd}\ t)^{CC} = \mathrm{snd}\ t^{CC} \end{aligned}$$

$$\begin{aligned} (\mathrm{inl}\ t)^{CC} &= \mathrm{inl}\ t^{CC} \\ (\mathrm{inr}\ t)^{CC} &= \mathrm{inr}\ t^{CC} \end{aligned} \qquad \begin{pmatrix} \mathbf{case}\ s\ \mathbf{of}\ \mathrm{inl}\ x \to t_1 \\ \qquad\qquad\ \mathrm{inr}\ y \to t_2 \end{pmatrix}^{CC} = \begin{pmatrix} \mathbf{case}\ s^{CC}\ \mathbf{of}\ \mathrm{inl}\ x \to t_1^{CC} \\ \qquad\qquad\ \mathrm{inr}\ y \to t_2^{CC} \end{pmatrix}$$

$$r^{CC} = r \qquad (\mathrm{sign}\ t)^{CC} = \mathrm{sign}\ t^{CC} \qquad (\mathrm{op}_{\mathbb{R}}(t_1, \ldots, t_n))^{CC} = \mathrm{op}_{\mathbb{R}}(t_1^{CC}, \ldots, t_n^{CC})$$

[34] We formulate closure conversion using a tagged sum type, rather than an untyped existential type as is sometimes done. The motivation is that we need to use equality checks on type tags at runtime for the casts and addition in the CHAD transformation.

Then, $\Gamma \vdash t : \tau$ implies $\Gamma^{CC} \vdash t^{CC} : \tau^{CC}$, where we define $(x_1 : \tau_1, \ldots, x_n : \tau_n)^{CC}$ as $x_1 : \tau_1^{CC}, \ldots, x_n : \tau_n^{CC}$.

Crucially, t^{CC} computes the same function[35] as t and does so with a proportional number of computation steps. In fact, most functional languages compile function types via closure conversion.

After applying closure conversion, we can apply CHAD as follows to types without free type variables (monomorphic types) as well as their programs:

$$
\begin{aligned}
&\mathcal{D}[\sigma \xrightarrow{\square} \tau]_1 = \mathcal{D}[\sigma]_1 \xrightarrow{\square} (\mathcal{D}[\tau]_1 \times (\mathcal{D}[\tau]_2 \to \mathcal{D}[\sigma]_2))\\
&\mathcal{D}[\sigma \xrightarrow{\square} \tau]_2 = \mathbf{1}\\
&\mathcal{D}[\Sigma_{\alpha:\mathsf{Type}}\ \tau]_1 = \Sigma_{\alpha:\mathsf{Type}}\ \mathcal{D}[\tau]_1\\
&\mathcal{D}[\Sigma_{\alpha:\mathsf{Type}}\ \tau]_2 = \underline{\Sigma}_{\alpha:\mathsf{Type}}\ \mathcal{D}[\tau]_2\\
&\mathcal{D}_\Gamma[\lambda(x:\tau).\ t] = \langle \lambda x : \mathcal{D}[\tau]_1.\ \mathcal{D}_{x:\tau}[t], \lambda\langle\rangle.\ \underline{0}\rangle\\
&\mathcal{D}_\Gamma[s\ t] = \mathbf{let}\ \langle f, \_\rangle = \mathcal{D}_\Gamma[s]\\
&\qquad\qquad\qquad \langle x, x'\rangle = \mathcal{D}_\Gamma[t]\\
&\qquad\qquad\qquad \langle y, y'\rangle = f\ x\\
&\qquad\qquad\ \mathbf{in}\ \langle y, \lambda d.\ x'\ (y'\ d)\rangle\\
&\mathcal{D}_\Gamma[\mathbf{pack}_\rho\ t] = \mathbf{let}\ \langle x, x'\rangle = \mathcal{D}_\Gamma[t]\\
&\qquad\qquad\qquad\ \mathbf{in}\ \langle \mathbf{pack}_\rho\ x, \lambda d.\ x'\ (\mathbf{cast}_\rho\ d)\rangle\\
&\mathcal{D}_\Gamma[\mathbf{case}\ t\ \mathbf{of}\ \mathbf{pack}_\alpha\ x \to s] = \mathbf{let}\ \langle z, z'\rangle = \mathcal{D}_\Gamma[t]\\
&\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{in\ case}\ z\ \mathbf{of}\ \mathbf{pack}_\alpha\ x \to\\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathbf{let}\ \langle y, y'\rangle = \mathcal{D}_\Gamma[s]\\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathbf{in}\ \langle y, \lambda v.\ \mathbf{let}\ \langle w_1, w_2\rangle = \mathbf{split}\ (y'\ v)\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{in}\ w_1 + z'\ (\mathbf{pack}_\rho\ w_2)\rangle
\end{aligned}
$$

Here, $\underline{\Sigma}_{\alpha:\mathsf{Type}}\ \tau$ has the following API:

$$
\begin{array}{ll}
\mathbf{pack}_\sigma : \tau[\sigma/\alpha] \to \underline{\Sigma}_{\alpha:\mathsf{Type}}\ \tau & \mathbf{cast}_\sigma : (\underline{\Sigma}_{\alpha:\mathsf{Type}}\ \tau) \to \tau[\sigma/\alpha]\\
\underline{0}_{\underline{\Sigma}_{\alpha:\mathsf{Type}}\ \tau} : \underline{\Sigma}_{\alpha:\mathsf{Type}}\ \tau & (+_{\underline{\Sigma}_{\alpha:\mathsf{Type}}\ \tau}) : (\underline{\Sigma}_{\alpha:\mathsf{Type}}\ \tau) \times (\underline{\Sigma}_{\alpha:\mathsf{Type}}\ \tau) \to \underline{\Sigma}_{\alpha:\mathsf{Type}}\ \tau
\end{array}
$$

which we can implement, completely analogously to the case of binary sum types, by representing $\underline{\Sigma}_{\alpha:\mathsf{Type}}\ \tau$ as $\mathbf{1} \sqcup (\Sigma_{\alpha:\mathsf{Type}}\ \tau)$. To implement this API, it is crucial that we work with Σ-types that hold type tags with decidable equality rather than untagged existentials. Observe that similarly to our treatment of sum types (with **lcastl** and **lcastr**), the differentiation of existential types requires some runtime casting,[36] in the absence of a type system with full dependent types. We can prove, however, that all required casts are type-safe in a stronger dependently-typed type system.

[35] Indeed, for any program t between first order types (types on which $[-]^{CC}$ acts as the identity), t is $\beta\eta$-equal to t^{CC}, so in particular is observationally equivalent.

[36] This throws an error if it fails — such an error will never be hit by the code generated by CHAD after closure conversion.

If we are willing to do a global program analysis, we can identify at compile-time the finite subset of components of the sum types that are actually used in practice. This allows us to simply replace the infinite sum type with a finite sum type (a transformation that we referred to as "deexistentialisation" in Section 6.8). We have now effectively arrived at our combination of closure conversion and defunctionalisation that we discussed in Section 6.8.

6.10.3 Cost model

In the table below, we describe the cost model (using call-by-value evaluation) used in the formalised proof in natural language. In the Agda formalisation (Section 6.10.4), this is embedded in the `eval` function, namely in its second component, as well as (for the **EVM** methods) in their implementation in the `spec.LACM` module. We separately describe the model here to aid in understanding what is encoded in the formal specification.

As in the rest of the chapter, we use $\mathrm{cost}(t;\Gamma)$ to denote the cost of evaluating t in the evaluation environment Γ. We furthermore use $\mathrm{eval}(t;\Gamma)$ to denote the *result* of evaluating t in that evaluation environment. $FV(t)$ denotes the set of free variables of the term t (only used for lambda abstraction to measure the size of the closure to allocate).

The term language that we analyse is `Term`, in `spec.agda`.

Term t	**Cost:** $\mathrm{cost}(t;\Gamma)$
x *(variable)*	1
let $x = s$ **in** t	$1 + \mathrm{cost}(s;\Gamma) + \mathrm{cost}(t; x = \mathrm{eval}(s;\Gamma), \Gamma)$
$\lambda x.\ t$	$1 + \lvert FV(\lambda x.\ t)\rvert$
$s\ t$	$1 + \mathrm{cost}(s;\Gamma) + \mathrm{cost}(t;\Gamma)$ $+ \mathrm{cost}(f\ x; f = \mathrm{eval}(s;\Gamma), x = \mathrm{eval}(t;\Gamma), \Gamma) - 2$ *(The -2 compensates for the two superfluous variable references f and x.)*
$\mathrm{op}_{\mathbb{R}}(t)$	$1 + \mathrm{cost}(t;\Gamma)$ *(for simplicity we assume unary operators only; n-ary operators take tuples, e.g. $(+) : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$)*
$\langle\rangle$	1
$\langle s, t\rangle$	$1 + \mathrm{cost}(s;\Gamma) + \mathrm{cost}(t;\Gamma)$
$\mathrm{fst}\ t$	$1 + \mathrm{cost}(t;\Gamma)$
$\mathrm{snd}\ t$	$1 + \mathrm{cost}(t;\Gamma)$
$\mathrm{inl}\ t$	$1 + \mathrm{cost}(t;\Gamma)$
$\mathrm{inr}\ t$	$1 + \mathrm{cost}(t;\Gamma)$
case s **of**	**case** $\mathrm{eval}(s;\Gamma)$ **of**
$\quad \mathrm{inl}\ x \to t_1$	$\quad \mathrm{inl}\ x' \to 1 + \mathrm{cost}(s;\Gamma) + \mathrm{cost}(t_1; x = x', \Gamma)$
$\quad \mathrm{inr}\ y \to t_2$	$\quad \mathrm{inr}\ y' \to 1 + \mathrm{cost}(s;\Gamma) + \mathrm{cost}(t_2; y = y', \Gamma)$

Because the implementations of the cotangent monoids are kept abstract in the term language used in the formalisation (the definitions can be found in the `spec.linear-types` module), `Term` has separate constructors for them and thus they get a separate treatment in the cost model. To understand the costs here, refer to the semantics of these operations given in Section 6.3.1 ($\sigma \underline{\times} \tau$) and Fig. 6.4 ($\sigma \underline{\sqcup} \tau$, which was unchanged in Section 6.3). If you are reading along with `eval` in the formalisation, note that `snd (zerov` τ`)` is always 1 currently; this '1' is inlined in the costs in the table below.

Term t	**Cost:** $\mathrm{cost}(t;\Gamma)$
$\langle\!\rangle$	1
$\langle s, t\rangle$	$1 + \mathrm{cost}(s;\Gamma) + \mathrm{cost}(t;\Gamma)$
lfst t	**case** $\mathrm{eval}(t;\Gamma)$ **of**
	nothing $\to 1 + \mathrm{cost}(t;\Gamma) + 1$ *(additional '1' to compute the* $\underline{0}$*)*
	just _ $\to 1 + \mathrm{cost}(t;\Gamma)$
lsnd t	**case** $\mathrm{eval}(t;\Gamma)$ **of**
	nothing $\to 1 + \mathrm{cost}(t;\Gamma) + 1$ *(idem)*
	just _ $\to 1 + \mathrm{cost}(t;\Gamma)$
linl t	$1 + \mathrm{cost}(t;\Gamma)$
linr t	$1 + \mathrm{cost}(t;\Gamma)$
lcastl t	**case** $\mathrm{eval}(t;\Gamma)$ **of**
	nothing $\to 1 + \mathrm{cost}(t;\Gamma) + 1$ *(idem)*
	just $x \to$ **case** x **of** { inl _ $\to 1 + \mathrm{cost}(t;\Gamma)$ | inr _ $\to$ **error** }
lcastr t	**case** $\mathrm{eval}(t;\Gamma)$ **of**
	nothing $\to 1 + \mathrm{cost}(t;\Gamma) + 1$ *(idem)*
	just $x \to$ **case** x **of** { inl _ $\to$ **error** | inr _ $\to 1 + \mathrm{cost}(t;\Gamma)$ }
$\underline{0}_{\sigma\underline{\times}\tau}$	1
$\underline{0}_{\sigma\underline{\sqcup}\tau}$	1

As for $\underline{0}_{\underline{\mathbb{R}}}$: the formalisation defines a primitive operation `LZERO` : $\underline{\mathbf{1}} \to \underline{\mathbb{R}}$, hence computing $\underline{0}_{\underline{\mathbb{R}}}$ takes $\mathrm{cost}(\texttt{LZERO}(\langle\!\rangle);\Gamma) = 1 + 1 = 2$ steps. The choice for this design was fairly arbitrary.

Finally, for the local accumulation monad (**EVM**; the specific implementation in Agda is called `LACM`), the situation is slightly more complex because the cost of a particular monadic computation depends on the incoming derivative vector, which is not known at the point where the methods are invoked. Thus the Agda code splits the cost model in two parts:

1. The implementation of `LACM` tracks the number of steps taken within the monad methods, as well as in the continuation of ($\gg\!=$). This is possible because we modified the type of ($\gg\!=$) (`bind`) as follows, abridged from the

Agda code:

$$\text{bind} : \texttt{LACM}\ \Gamma\ \sigma \to (\sigma \to \texttt{LACM}\ \Gamma\ \tau \times \mathbb{Z}) \to \texttt{LACM}\ \Gamma\ \tau$$

That is to say, the continuation additionally returns the number of steps taken therein. This accumulated total number of steps is finally returned as the cost of calling **run** on the whole computation. The intended semantics is that the monadic computation does not actually run until, well, **run**ning it, at which point the computation runs to completion, collecting the number of steps taken, which then gets returned.

2. The evaluator does not know anything about the internals of the monad, and simply accounts constant cost for adding the operation in question to the pending computation in memory — except for **run**, where of course the returned cost is added to the total.

The costs in point (1) are available to the proof through a list of *properties* about the monad; the actual monad implementation is hidden through the use of an `abstract` block. The types of the monad methods as the proof sees them are as follows:

$$\begin{aligned}
\texttt{pure} &: \tau \to \texttt{LACM}\ \Gamma\ \tau \\
\texttt{bind} &: \texttt{LACM}\ \Gamma\ \sigma \to (\sigma \to \texttt{LACM}\ \Gamma\ \tau \times \mathbb{Z}) \to \texttt{LACM}\ \Gamma\ \tau \\
\texttt{run} &: \texttt{LACM}\ \Gamma\ \tau \to \overline{\Gamma} \to \tau \times (\overline{\Gamma} \times \mathbb{Z}) \\
\texttt{add} &: \texttt{Idx}\ \Gamma\ \tau \to \tau \to \texttt{LACM}\ \Gamma\ \mathbf{1} \\
\texttt{scope} &: \tau \to \texttt{LACM}\ (\tau :: \Gamma)\ \sigma \to \texttt{LACM}\ \Gamma\ (\tau \times \sigma)
\end{aligned}$$

The mentioned properties can be found in the `spec.LACM` module in Section 6.10.4.

The costs accounted by `eval` are as follows, unsurprising as usual:

Term t	**Cost:** $\text{cost}(t;\Gamma)$
return t	$1 + \text{cost}(t;\Gamma)$
$s \gg\!= t$	$1 + \text{cost}(s;\Gamma) + \text{cost}(t;\Gamma)$
run s t	$1 + \text{cost}(s;\Gamma) + \text{cost}(t;\Gamma)$ $+ \text{snd}\ (\text{snd}\ (\texttt{run}\ (\text{eval}(s;\Gamma))\ (\text{eval}(t;\Gamma))))$
one t	$1 + \text{cost}(t;\Gamma)$
scope s t	$1 + \text{cost}(s;\Gamma) + \text{cost}(t;\Gamma)$

6.10.4 Agda formalisation specification

The specification of the Agda proof, which provides only those definitions necessary to state the main complexity theorems (but not prove them), follows on

subsequent pages. This specification consists of three modules; each module starts on a new page.

The full formalisation can be found at `https://github.com/tomsmeding/efficient-chad-agda`.

```
module spec.linear-types where

open import Agda.Builtin.Float using (Float; primFloatPlus)
open import Agda.Builtin.Maybe using (Maybe; nothing; just)
open import Agda.Builtin.Sigma using (_,_; fst; snd)
open import Agda.Builtin.Unit using (⊤; tt)

open import Data.List using (List; []; _∷_)
open import Data.Integer using (ℤ; _+_; +_)
open import Data.Product using (_×_)
open import Data.Sum using (_⊎_; inj₁; inj₂)


-- The linear (i.e. monoidal) types. These types have a monoid structure, and
-- have a potential function (φ) defined on them.
data LTyp : Set where
  LUn LR : LTyp
  _:*!_ : LTyp -> LTyp -> LTyp
  _:+!_ : LTyp -> LTyp -> LTyp

-- A linear typing environment is a list of linear types.
LEnv : Set
LEnv = List LTyp

-- The representation (semantics) of the linear types; the representation of
-- normal types follows in `spec.agda`.
LinRep : LTyp -> Set
LinRep LUn = ⊤
LinRep LR = Float
LinRep (σ :*! τ) = Maybe (LinRep σ × LinRep τ)
LinRep (σ :+! τ) = Maybe (LinRep σ ⊎ LinRep τ)

-- Linear environment tuple: a tuple of all the types in a linear environment.
-- This is used to pass a linear environment as a _value_ into, and out of,
-- the monadic computation in the target program.
LEtup : LEnv -> Set
LEtup [] = ⊤
LEtup (τ ∷ Γ) = LinRep τ × LEtup Γ

-- An index into a typing environment
data Idx {n} {typ : Set n} : List typ -> typ -> Set n where
  Z : {e : List typ} {τ : typ} -> Idx (τ ∷ e) τ
  S : {e : List typ} {τ τ' : typ} -> Idx e τ -> Idx (τ' ∷ e) τ

one : ℤ
one = + 1

-- The zero part of the monoid structure of the linear types. Aside from
-- returning the value, this also returns an integer recording the number of
-- evaluation steps taken during the operation. This integer is used for
-- complexity analysis.
-- Because zerov and plusv are not implemented in terms of evaluation of terms,
-- we simply use an approximation here that is proportional to the actual
-- number of steps. In practice this means that we can take c_φ = 1.
zerov : (τ : LTyp) -> LinRep τ × ℤ
zerov LUn = tt , one
zerov LR = 0.0 , one
zerov (σ :*! τ) = nothing , one
zerov (σ :+! τ) = nothing , one

-- The addition part of the monoid structure of the linear types. Similarly,
```

```
-- the number of evaluation steps is returned.
--
-- For sum types, we return zero on adding incompatible values instead of
-- throwing an error. This prevents D2τ (σ :+ τ) from being a monoid, but of
-- course, if a proper implementation that would throw errors does not in fact
-- error on a given input program, the implementation here would not introduce
-- values that violate the monoid laws either.
--
-- In particular, because the dependently-typed variant of CHAD is correct (see
-- Nunes, Vákár. "CHAD for expressive total languages." MSCS 2023), there is an
-- external proof that those error cases would be impossible, and thus that the
-- cases that violate the monoid laws here are also impossible.
--
-- We put up with this infelicity because it allows us to avoid having to model
-- partiality in our language, which is no fundamental issue but introduces a
-- large amount of administration everywhere that makes the proof harder to
-- read and to write.
plusv : (τ : LTyp) -> LinRep τ -> LinRep τ -> LinRep τ × ℤ
plusv LUn tt tt = tt , one
plusv LR x y = primFloatPlus x y , one
plusv (σ :*! τ) nothing y = y , one
plusv (σ :*! τ) x nothing = x , one
plusv (σ :*! τ) (just (x , y)) (just (x' , y')) =
  let xr , cx = plusv σ x x'
      yr , cy = plusv τ y y'
  in just (xr , yr) , one + cx + cy
plusv (σ :+! τ) x nothing = x , one
plusv (σ :+! τ) nothing y = y , one
plusv (σ :+! τ) (just (inj₁ x)) (just (inj₁ y)) =
  let z , cz = plusv σ x y
  in just (inj₁ z) , one + cz
plusv (σ :+! τ) (just (inj₂ x)) (just (inj₂ y)) =
  let z , cz = plusv τ x y
  in just (inj₂ z) , one + cz
plusv (σ :+! τ) _ _ = nothing , one  -- NOTE: proper implementation would error

-- Add the value 'val' into the position 'idx' in the environment tuple.
addLEτ : {Γ : LEnv} {τ : LTyp}
      -> (idx : Idx Γ τ) -> (val : LinRep τ) -> LEtup Γ -> LEtup Γ
addLEτ Z val (x , env) = fst (plusv _ val x) , env
addLEτ (S i) val (x , env) = x , addLEτ i val env

-- Project a value out of an environment tuple.
_Eτ!!_ : {Γ : LEnv} {τ : LTyp} -> LEtup Γ -> Idx Γ τ -> LinRep τ
(x , env) Eτ!! Z = x
(x , env) Eτ!! (S i) = env Eτ!! i
```

```
module spec.LACM where

open import Agda.Builtin.Sigma using (_,_; fst; snd)
open import Agda.Builtin.Unit using (⊤; tt)

open import Data.Integer using (ℤ; _+_; +_; _-_)
open import Data.List using (_∷_; length)
open import Data.Product using (_×_)
open import Relation.Binary.PropositionalEquality using (_≡_; refl; cong; trans)

open import Data.Integer.Solver using (module +-*-Solver)
open +-*-Solver using (solve; _:+_; _:-_; :-_; con; _:=_)

open import spec.linear-types


-- In Agda, an 'abstract' block prevents the contained definitions from
-- _reducing_ outside of the block. In effect, this means that outside of the
-- 'abstract' block, only the type signatures of its definitions are visible,
-- not the bodies. We use this to ensure that the complexity proof depends only
-- on the semantics and the complexities of the LACM interpretation, not its
-- actual implementation.
abstract
  -- Local accumulation monad.
  LACM : LEnv -> Set -> Set
  LACM Γ a = LEtup Γ -> a × LEtup Γ × ℤ

  -- The methods of the monad, including pure and bind.

  pure : ∀ {Γ : LEnv} {a : Set} -> a -> LACM Γ a
  pure x e = x , e , one

  -- Note that the continuation should also return the cost of evaluating the
  -- continuation; this cost will be included in the cost returned by 'run'
  -- when handling the top-level monadic computation.
  bind : ∀ {Γ : LEnv} {a b : Set} -> LACM Γ a -> (a -> LACM Γ b × ℤ) -> LACM Γ b
  bind f g e =
    let x , e1 , c1 = f e
        m2 , ccall = g x
        y , e2 , c2 = m2 e1
    in y , e2 , one + c1 + ccall + c2

  -- Returns computation result, the output environment, and the cost of
  -- evaluating the monadic computation.
  run : ∀ {Γ : LEnv} {a : Set} -> LACM Γ a -> LEtup Γ -> a × LEtup Γ × ℤ
  run {Γ} f e =
    let r , e' , c = f e
    in r , e' , one + + length Γ + c

  -- Add the given value to the value at the given index in the state.
  add : ∀ {Γ : LEnv} {τ : LTyp} -> Idx Γ τ -> LinRep τ -> LACM Γ ⊤
  add {τ = τ} Z x (y , e) =
    let z , cz = plusv τ x y
    in tt , (z , e) , one + cz
  add (S i) x (y , e) =          -- supposed to be O(1) access, so we don't cost
    let r , e' , c = add i x e   -- this traversal
    in r , (y , e') , c

  -- Temporarily add a new cell to the state. The final value of this cell is
  -- returned inside the monad when 'scope' finishes.
  scope : ∀ {Γ : LEnv} {τ : LTyp} {a : Set}
```

```
      -> LinRep τ -> LACM (τ :: Γ) a -> LACM Γ (LinRep τ × a)
scope x f e =
  let r , (x' , e') , c = f (x , e)
  in (x' , r) , e' , one + c

-- Properties about the monad methods that we need when reasoning about LACM
-- in the complexity proof. These four lemmas "push" 'run' down inside
-- 'pure', 'bind', 'add' and 'scope', and thereby define properties both
-- about the _semantics_ of the monad, as well as its _complexity_.
--
-- The cost counts are used abstractly here: a cost of 1 indicates some O(1)
-- work, not necessarily one single step in the underlying machine. Or,
-- alternatively: it is assumed that the underlying machine has support for
-- these (side-effectful) operations and can execute them in the indicated
-- number of "steps". Ultimately, this matters little, because this proof is
-- only concerned with complexity, not absolute performance (which would need
-- to be benchmarked on an actual machine anyway).

run-pure : ∀ {Γ : LEnv} {a : Set} -> (x : a)
        -> (env : LEtup Γ)
        -> let _ , env' , c = run {Γ} (pure x) env
           in (env' ≡ env) × (c ≡ one + + length Γ + one)

run-bind : ∀ {Γ : LEnv} {a b : Set}
        -> (m1 : LACM Γ a) -> (k : a -> LACM Γ b × ℤ)
        -> (env : LEtup Γ)
        -> let _ , env' , c = run (bind m1 k) env
               r1 , env1 , c1 = run m1 env
               m2 , ccall = k r1
               r2 , env2 , c2 = run m2 env1
           in (env' ≡ env2) × (c ≡ c1 + ccall + c2 - + length Γ)

run-add : ∀ {Γ : LEnv} {τ : LTyp}
       -> (idx : Idx Γ τ) -> (val : LinRep τ)
       -> (env : LEtup Γ)
       -> let tt , env' , c = run (add idx val) env
          in (env' ≡ addLEτ idx val env)
             × (c ≡ + 2 + snd (plusv τ val (env Eτ!! idx)) + + length Γ)

run-scope : ∀ {Γ : LEnv} {a : Set} {τ : LTyp}
         -> (m : LACM (τ :: Γ) a) -> (inval : LinRep τ)
         -> (env : LEtup Γ)
         -> let (outval1 , x1) , env1 , c1 = run (scope inval m) env
                x2 , (outval2 , env2) , c2 = run m (inval , env)
            in (x1 ≡ x2) × (outval1 ≡ outval2) × (env1 ≡ env2) × (c1 ≡ c2)

-- Proofs for the above properties; these are elided in the appendix.
```

```
module spec where

open import Agda.Builtin.Bool using (true; false)
open import Agda.Builtin.Float
  using (Float; primFloatPlus; primFloatTimes; primFloatNegate; primFloatLess)
open import Agda.Builtin.Maybe using (nothing; just)
open import Agda.Builtin.Sigma using (_,_; fst; snd)
open import Agda.Builtin.Unit using (⊤; tt)

open import Data.Nat using (ℕ) renaming (_+_ to _+ℕ_)
open import Data.Fin using (Fin; zero; suc)
open import Data.List using (List; []; _∷_; length; map)
open import Data.Integer using (ℤ; _+_; _-_; _*_; -_; +_; _≤_)
open import Data.Product using (_×_)
open import Data.Sum using (_⊎_; inj₁; inj₂)
open import Function.Base using (id; _$_; _∘_; case_of_)
open import Relation.Binary.PropositionalEquality
              using (_≡_; refl; sym; subst; cong)

open import spec.linear-types public
import spec.LACM as LACM
open LACM using (LACM)


-------------------- UTILITY FUNCTIONS ----------------------------------------

-- Project from a list with a bounded index into that list. Not sure why this
-- is not in the standard library for lists.
_!!_ : ∀ {n} {a : Set n} -> (l : List a) -> Fin (length l) -> a
(x ∷ xs) !! zero = x
(x ∷ xs) !! suc i = xs !! i


-------------------- TYPES ----------------------------------------------------
-- Linear types were already defined in spec.linear-types; here are the
-- regular, non-linear types.

-- Types are indexed by whether they are primal types (i.e. types that occur in
-- the source program) or dual types (i.e. types that occur in the target
-- program).
data PDTag : Set where
  Pr Du : PDTag

-- The types of the object language. The source language is typed by
-- 'Typ Pr', which allows only a few types. The target language is typed by
-- 'Typ Du', which allows not only the source language types but also a couple
-- of other types, including functions and the linear types.
data Typ : PDTag -> Set where
  Un Inte R : ∀ {tag} -> Typ tag
  _:*_ : ∀ {tag} -> Typ tag -> Typ tag -> Typ tag
  _:+_ : ∀ {tag} -> Typ tag -> Typ tag -> Typ tag

  _:->_ : Typ Du -> Typ Du -> Typ Du
  -- Environment vector monad. This is the same EVM as in the paper; the
  -- implementation is LACM.
  EVM : LEnv -> Typ Du -> Typ Du

  -- The linear types (with embedded potential)
  Lin : LTyp -> Typ Du

-- A normal typing environment is a list of types.
```

```
Env : PDTag -> Set
Env tag = List (Typ tag)

-- The representation / semantics of our types. LinRep from spec.linear-types
-- does the same for the linear types LTyp.
Rep : ∀ {tag} -> Typ tag -> Set
Rep Un = ⊤
Rep Inte = ℤ
Rep R = Float
Rep (σ :* τ) = Rep σ × Rep τ
Rep (σ :+ τ) = Rep σ ⊎ Rep τ
Rep (σ :-> τ) = Rep σ -> Rep τ × ℤ
Rep (EVM Γ τ) = LACM Γ (Rep τ)
Rep (Lin τ) = LinRep τ

-- Convert a type from a source-language type to a target-language type. This
-- function is operationally the identity.
dut : Typ Pr -> Typ Du
dut Un = Un
dut Inte = Inte
dut R = R
dut (σ :* τ) = dut σ :* dut τ
dut (σ :+ τ) = dut σ :+ dut τ

-- The embedded counterpart of LEtup: a tuple of all the types in a linear
-- environment. This is used to pass a linear environment as a _value_ into,
-- and out of, the monadic computation in the target program.
LEτ : LEnv -> Typ Du
LEτ [] = Un
LEτ (τ :: Γ) = Lin τ :* LEτ Γ

-- LEtup and ⟦LEτ⟧ are the same thing.
LEtup-eq-LEτ : (Γ : LEnv) -> Rep (LEτ Γ) ≡ LEtup Γ
LEtup-eq-LEτ [] = refl
LEtup-eq-LEτ (τ :: Γ) rewrite LEtup-eq-LEτ Γ = refl

LEtup-to-LEτ : (Γ : LEnv) -> Rep (LEτ Γ) -> LEtup Γ
LEtup-to-LEτ [] x = x
LEtup-to-LEτ (τ :: Γ) (x , env) = x , LEtup-to-LEτ Γ env

LEτ-to-LEtup : (Γ : LEnv) -> LEtup Γ -> Rep (LEτ Γ)
LEτ-to-LEtup [] x = x
LEτ-to-LEtup (τ :: Γ) (x , env) = x , LEτ-to-LEtup Γ env


-------------------- PRIMITIVE OPERATIONS -------------------------------------

-- The primitive operations in our object language. Again, some operations are
-- available both in the source and in the target language, whereas others (the
-- 'Du'-indexed ones) are available only in the target language.
data Primop : (tag : PDTag) -> (σ τ : Typ tag) -> Set where
  ADD  : ∀ {tag} -> Primop tag (R :* R) R
  MUL  : ∀ {tag} -> Primop tag (R :* R) R
  NEG  : ∀ {tag} -> Primop tag R R
  LIT  : ∀ {tag} -> Float -> Primop tag Un R
  IADD : ∀ {tag} -> Primop tag (Inte :* Inte) Inte
  IMUL : ∀ {tag} -> Primop tag (Inte :* Inte) Inte
  INEG : ∀ {tag} -> Primop tag Inte Inte
  -- sign: (negative or positive) or zero/NaN
  SIGN : ∀ {tag} -> Primop tag R ((Un :+ Un) :+ Un)

  LZERO  : Primop Du (Lin LUn) (Lin LR)
```

```
  LADD   : Primop Du (Lin LR :* Lin LR) (Lin LR)
  LSCALE : Primop Du (Lin LR :* R) (Lin LR)
  LNEG   : Primop Du (Lin LR) (Lin LR)

-- Semantics of the primitive operations.
evalprim : ∀ {tag} {σ τ} -> Primop tag σ τ -> Rep σ -> Rep τ
evalprim ADD (x , y) = primFloatPlus x y
evalprim MUL (x , y) = primFloatTimes x y
evalprim NEG x = primFloatNegate x
evalprim (LIT x) tt = x
evalprim IADD (x , y) = x + y
evalprim IMUL (x , y) = x * y
evalprim INEG x = - x
evalprim SIGN x =
  case primFloatLess x 0.0 of
    λ where true  -> inj₁ (inj₁ tt)
            false -> case primFloatLess 0.0 x of
                       λ where true  -> inj₁ (inj₂ tt)
                               false -> inj₂ tt
evalprim LZERO tt = 0.0
evalprim LADD (x , y) = primFloatPlus x y
evalprim LSCALE (x , y) = primFloatTimes x y
evalprim LNEG x = primFloatNegate x


-------------------- OBJECT LANGUAGE --------------------------------------------

-- The object language. The source language and the target language are both
-- expressed using the same Term data type, just with a different index: a
-- source term is of type 'Term Pr Γ τ', whereas a target term is of type
-- 'Term Du Γ τ'.
data Term : (tag : PDTag) -> (Γ : Env tag) -> (τ : Typ tag) -> Set where
  var   : ∀ {tag} {Γ : Env tag} {τ : Typ tag}
        -> Idx Γ τ -> Term tag Γ τ
  let'  : ∀ {tag} {Γ : Env tag} {σ τ : Typ tag}
        -> Term tag Γ σ -> Term tag (σ :: Γ) τ -> Term tag Γ τ

  prim  : ∀ {tag} {Γ : Env tag} {σ τ : Typ tag}
        -> Primop tag σ τ -> Term tag Γ σ -> Term tag Γ τ

  unit  : ∀ {tag} {Γ : Env tag}
        -> Term tag Γ Un

  pair  : ∀ {tag} {Γ : Env tag} {σ τ : Typ tag}
        -> Term tag Γ σ -> Term tag Γ τ -> Term tag Γ (σ :* τ)
  fst'  : ∀ {tag} {Γ : Env tag} {σ τ : Typ tag}
        -> Term tag Γ (σ :* τ) -> Term tag Γ σ
  snd'  : ∀ {tag} {Γ : Env tag} {σ τ : Typ tag}
        -> Term tag Γ (σ :* τ) -> Term tag Γ τ

  inl   : ∀ {tag} {Γ : Env tag} {σ τ : Typ tag}
        -> Term tag Γ σ -> Term tag Γ (σ :+ τ)
  inr   : ∀ {tag} {Γ : Env tag} {σ τ : Typ tag}
        -> Term tag Γ τ -> Term tag Γ (σ :+ τ)
  case' : ∀ {tag} {Γ : Env tag} {σ τ ρ : Typ tag}
        -> Term tag Γ (σ :+ τ)
        -> Term tag (σ :: Γ) ρ -> Term tag (τ :: Γ) ρ
        -> Term tag Γ ρ

  -- The Γ' is the closure of the lambda. We model this explicitly because the
  -- cost of evaluating 'lam' is linear in the size of its closure, so it is
  -- worth keeping it small.
```

```
  lam   : {Γ : Env Du} {τ : Typ Du}
        -> {σ : Typ Du}     -- Γ' is a subset of Γ
        -> (Γ' : Env Du) -> ({ρ : Typ Du} -> Idx Γ' ρ -> Idx Γ ρ)
        -> Term Du (σ :: Γ') τ -> Term Du Γ (σ :-> τ)
  app   : {Γ : Env Du} {σ τ : Typ Du}
        -> Term Du Γ (σ :-> τ) -> Term Du Γ σ -> Term Du Γ τ

  pureevm : {Γ : Env Du} {Γ' : LEnv} {τ : Typ Du}
          -> Term Du Γ τ -> Term Du Γ (EVM Γ' τ)
  bindevm : {Γ : Env Du} {Γ' : LEnv} {σ τ : Typ Du}
          -> Term Du Γ (EVM Γ' σ) -> Term Du Γ (σ :-> EVM Γ' τ)
          -> Term Du Γ (EVM Γ' τ)
  runevm : {Γ : Env Du} {Γ' : LEnv} {τ : Typ Du}
         -> Term Du Γ (EVM Γ' τ) -> Term Du Γ (LEτ Γ') -> Term Du Γ (τ :* LEτ Γ')
  addevm : {Γ : Env Du} {Γ' : LEnv} {τ : LTyp}
         -> Idx Γ' τ -> Term Du Γ (Lin τ) -> Term Du Γ (EVM Γ' Un)
  scopeevm : {Γ : Env Du} {Γ' : LEnv} {τ : LTyp} {σ : Typ Du}
           -> Term Du Γ (Lin τ) -> Term Du Γ (EVM (τ :: Γ') σ)
           -> Term Du Γ (EVM Γ' (Lin τ :* σ))

  lunit : {Γ : Env Du}
        -> Term Du Γ (Lin LUn)

  lpair : {Γ : Env Du} {σ τ : LTyp}
        -> Term Du Γ (Lin σ) -> Term Du Γ (Lin τ) -> Term Du Γ (Lin (σ :*! τ))
  lfst' : {Γ : Env Du} {σ τ : LTyp}
        -> Term Du Γ (Lin (σ :*! τ)) -> Term Du Γ (Lin σ)
  lsnd' : {Γ : Env Du} {σ τ : LTyp}
        -> Term Du Γ (Lin (σ :*! τ)) -> Term Du Γ (Lin τ)
  lpairzero : {Γ : Env Du} {σ τ : LTyp}
            -> Term Du Γ (Lin (σ :*! τ))

  linl : {Γ : Env Du} {σ τ : LTyp}
       -> Term Du Γ (Lin σ) -> Term Du Γ (Lin (σ :+! τ))
  linr : {Γ : Env Du} {σ τ : LTyp}
       -> Term Du Γ (Lin τ) -> Term Du Γ (Lin (σ :+! τ))
  lcastl : {Γ : Env Du} {σ τ : LTyp}
         -> Term Du Γ (Lin (σ :+! τ))
         -> Term Du Γ (Lin σ)
  lcastr : {Γ : Env Du} {σ τ : LTyp}
         -> Term Du Γ (Lin (σ :+! τ))
         -> Term Du Γ (Lin τ)
  lsumzero : {Γ : Env Du} {σ τ : LTyp}
           -> Term Du Γ (Lin (σ :+! τ))


-------------------- OBJECT LANGUAGE UTILITIES --------------------------------
-- Utilities for working with the object language: weakening and some
-- alternative forms of constructors.

-- A data type representing weakenings.
--
-- The reason we have this explicit representation of reindexing mappings, as
-- opposed to a general sink function with the following type:
--   sink : {Γ Γ' : Env tag} {τ : Typ tag}
--        -> ({σ : Typ tag} -> Idx Γ σ -> Idx Γ' σ)
--        -> Term tag Γ τ -> Term tag Γ' τ
-- is that with the above representation we'd need (a very weak form of?)
-- functional extensionality to use certain lemmas in the complexity proof. The
-- reason for that is that we'd like to use multiple lemmas about the same
-- things together, and all of those lemmas return facts about terms that
-- normalise to the same thing but contain uses of 'sink' applied to unknown
```

```
-- indices inside them. If 'sink' took a function argument, then proving
-- equality here would involve proving equality of functions given equal
-- syntactic representation, which Agda does not do, despite being much weaker
-- than full functional extensionality.
--
-- This 'Weakening' type does not model all such Idx->Idx functions, but since
-- we need only a very limited set of them and this data type is sufficient to
-- describe those, we can make our lives easy and work with this simple
-- representation.
data Weakening {tag} : (Γ Γ' : Env tag) -> Set where
  WEnd  : {Γ : Env tag} -> Weakening Γ Γ
  WCut  : {Γ' : Env tag} -> Weakening [] Γ'
  WCopy : {Γ Γ' : Env tag} {τ : Typ tag}
            -> Weakening Γ Γ' -> Weakening (τ :: Γ) (τ :: Γ')
  WSkip : {Γ Γ' : Env tag} {τ : Typ tag}
            -> Weakening Γ Γ' -> Weakening Γ (τ :: Γ')

-- Apply a weakening to a single index.
weaken-var
  : ∀ {tag} {Γ Γ' : Env tag}
 -> (w : Weakening Γ Γ')
 -> {τ : Typ tag}
 -> Idx Γ τ
 -> Idx Γ' τ
weaken-var WEnd i = i
weaken-var (WCopy w) Z = Z
weaken-var (WCopy w) (S i) = S (weaken-var w i)
weaken-var (WSkip w) i = S (weaken-var w i)

-- Sink a term using a weakening (an index remapping). A typical special case
-- is in 'sink1' below.
sink : ∀ {tag} {Γ Γ' : Env tag}
    -> {τ : Typ tag}
    -> Weakening Γ Γ'
    -> Term tag Γ τ
    -> Term tag Γ' τ
sink w (var i) = var (weaken-var w i)
sink w (let' e1 e2) = let' (sink w e1) (sink (WCopy w) e2)
sink w (lam Γ' inj e) = lam Γ' (weaken-var w ∘ inj) e
sink w (app e1 e2) = app (sink w e1) (sink w e2)
sink w (prim op e) = prim op (sink w e)
sink w unit = unit
sink w (pair e1 e2) = pair (sink w e1) (sink w e2)
sink w (fst' e) = fst' (sink w e)
sink w (snd' e) = snd' (sink w e)
sink w (inl e) = inl (sink w e)
sink w (inr e) = inr (sink w e)
sink w (case' e1 e2 e3) = case' (sink w e1)
                                (sink (WCopy w) e2) (sink (WCopy w) e3)
sink w (pureevm e) = pureevm (sink w e)
sink w (bindevm e1 e2) = bindevm (sink w e1) (sink w e2)
sink w (runevm e1 e2) = runevm (sink w e1) (sink w e2)
sink w (addevm i e) = addevm i (sink w e)
sink w (scopeevm e1 e2) = scopeevm (sink w e1) (sink w e2)
sink w lunit = lunit
sink w (lpair e1 e2) = lpair (sink w e1) (sink w e2)
sink w (lfst' e) = lfst' (sink w e)
sink w (lsnd' e) = lsnd' (sink w e)
sink w lpairzero = lpairzero
sink w (linl e) = linl (sink w e)
sink w (linr e) = linr (sink w e)
sink w (lcastl e) = lcastl (sink w e)
```

```
sink w (lcastr e) = lcastr (sink w e)
sink w lsumzero = lsumzero

-- Add one additional free variable to the bottom of the term's free variable
-- list (here of type σ). This, for example, allows one to put a term under one
-- additional let-binding (whose variable is unused in the term).
sink1 : ∀ {tag} {Γ : Env tag} {σ τ : Typ tag}
     -> Term tag Γ τ -> Term tag (σ :: Γ) τ
sink1 = sink (WSkip WEnd)

-- Build a closure. The 'lam' constructor in Term represents the inclusion of
-- the (smaller) closure environment into the larger containing environment
-- with an index remapping function, but writing those inline is cumbersome.
-- It's easier to simply give a list of indices into the containing environment
-- that you want to include in the closure. This 'lamwith' function allows you
-- to do that; said list is the list 'vars'. 'α' is the argument type of the
-- lambda.
lamwith : {α : Typ Du} {Γ : Env Du} {τ : Typ Du}
       -> (vars : List (Fin (length Γ)))
       -> Term Du (α :: map (\i -> Γ !! i) vars) τ
       -> Term Du Γ (α :-> τ)
lamwith {_} {Γ} vars body =
  lam (map (Γ !!_) vars)
      (buildinj vars)
      body
  where
    buildidx : {Γ : Env Du} -> (i : Fin (length Γ)) -> Idx Γ (Γ !! i)
    buildidx {[]} ()
    buildidx {_ :: _} zero = Z
    buildidx {_ :: _} (suc i) = S (buildidx i)

    buildinj : {Γ : Env Du} {ρ : Typ Du}
            -> (vars : List (Fin (length Γ)))
            -> Idx (map (\i -> Γ !! i) vars) ρ -> Idx Γ ρ
    buildinj (i :: vars) Z = buildidx i
    buildinj (i :: vars) (S idx) = buildinj vars idx

-- 'bindevm' from Term is '>>=' of the environment vector monad EVM; this is
-- '>>'. 'a >> b' is expanded to 'let x = b in a >>= \_ -> x'. Note the
-- creation of a closure using 'lamwith' containing one entry, namely x.
thenevm : {Γ : LEnv} {Γ' : Env Du}
      -> Term Du Γ' (EVM Γ Un) -> Term Du Γ' (EVM Γ Un) -> Term Du Γ' (EVM Γ Un)
thenevm a b =
  let' b $
    bindevm (sink1 a) (lamwith (zero :: []) (var (S Z)))

-- Generic index retyping utility. An index of type τ into an environment Γ can
-- be retyped as an index of modified type into a modified environment.
convIdx : ∀ {n} {typ typ' : Set n} {Γ : List typ} {τ : typ}
       -> (f : typ -> typ')
       -> Idx Γ τ -> Idx (map f Γ) (f τ)
convIdx f Z = Z
convIdx f (S i) = S (convIdx f i)


-------------------- DIFFERENTIATION --------------------------------------------
-- Derivative type mappings and derivatives of primitive operations.

-- The primal type mapping, written D[τ]₁ in the paper.
D1τ : Typ Pr -> Typ Du
D1τ Un = Un
D1τ Inte = Inte
```

```
D1τ R = R
D1τ (σ :* τ) = D1τ σ :* D1τ τ
D1τ (σ :+ τ) = D1τ σ :+ D1τ τ

-- The dual type mapping, written D[τ]₂ in the paper. Dual types are linear
-- (i.e. have a monoid structure).
D2τ' : Typ Pr -> LTyp
D2τ' Un = LUn
D2τ' Inte = LUn
D2τ' R = LR
D2τ' (σ :* τ) = D2τ' σ :*! D2τ' τ
D2τ' (σ :+ τ) = D2τ' σ :+! D2τ' τ

-- Dual type as a target language type.
D2τ : Typ Pr -> Typ Du
D2τ τ = Lin (D2τ' τ)

-- Primal environment mapping. This is D[Γ]₁ in the paper.
D1Γ : Env Pr -> Env Du
D1Γ = map D1τ

-- Dual environment mapping. Recall LEτ from above. This is \overline{D[Γ]₂} in
-- the paper.
D2Γtup : Env Pr -> Typ Du
D2Γtup Γ = LEτ (map D2τ' Γ)

-- The codomain of the backpropagator of a differentiated program. 'EVM' is the
-- environment vector monad, instantiated with the local accumulation monad
-- LACM. 'D2Γ' is used in the type of 'chad' below.
D2Γ : Env Pr -> Typ Du
D2Γ Γ = EVM (map D2τ' Γ) Un

-- Convert a _value_ of source-language type to a primal value in the
-- differentiated world. Because D1τ is the identity for non-function types,
-- this function is also the identity on values.
primal : (τ : Typ Pr) -> Rep τ -> Rep (D1τ τ)
primal Un tt = tt
primal Inte x = x
primal R x = x
primal (σ :* τ) (x , y) = primal σ x , primal τ y
primal (σ :+ τ) (inj₁ x) = inj₁ (primal σ x)
primal (σ :+ τ) (inj₂ y) = inj₂ (primal τ y)

-- Our primitive operations work on types of which the primal is the same as
-- the original type. This is of course true for _all_ our types in this Agda
-- development, but this ceases to be true once we add function types to the
-- source language. In that situation, we would thus require that primitive
-- operations do not take or return function values.
niceprim : {σ τ : Typ Pr} -> Primop Pr σ τ -> (D1τ σ ≡ dut σ) × (D1τ τ ≡ dut τ)
niceprim ADD = refl , refl
niceprim MUL = refl , refl
niceprim NEG = refl , refl
niceprim (LIT _) = refl , refl
niceprim SIGN = refl , refl
niceprim IADD = refl , refl
niceprim IMUL = refl , refl
niceprim INEG = refl , refl

-- The reverse derivative of a primitive operation. The returned term takes as
-- input (i.e. uses in its environment) the primal of its argument and the
-- cotangent of its output, and returns the cotangent of its argument. This is
-- wrapped in a more easily-used form below in 'dprim'.
```

```
dprim' : {σ τ : Typ Pr} -> Primop Pr σ τ -> Term Du (D2τ τ :: D1τ σ :: []) (D2τ σ)
dprim' ADD = lpair (var Z) (var Z)
dprim' MUL = lpair (prim LSCALE (pair (var Z) (snd' (var (S Z)))))
                   (prim LSCALE (pair (var Z) (fst' (var (S Z)))))
dprim' NEG = prim LNEG (var Z)
dprim' (LIT x) = lunit
dprim' SIGN = prim LZERO lunit
dprim' IADD = lpair lunit lunit
dprim' IMUL = lpair lunit lunit
dprim' INEG = lunit

-- More easy to use version of dprim' above, using let-bindings to take the two
-- input terms as separate arguments.
dprim : {Γ : Env Du} {σ τ : Typ Pr} -> Primop Pr σ τ
     -> Term Du Γ (D1τ σ) -> Term Du Γ (D2τ τ) -> Term Du Γ (D2τ σ)
dprim op p d =
  let' p $
  let' (sink1 d) $
    sink (WCopy (WCopy WCut)) (dprim' op)

-- Retype a source-language primitive operation as a target-language one.
duPrim : {σ τ : Typ Pr} -> Primop Pr σ τ -> Primop Du (dut σ) (dut τ)
duPrim ADD = ADD
duPrim MUL = MUL
duPrim NEG = NEG
duPrim (LIT x) = LIT x
duPrim SIGN = SIGN
duPrim IADD = IADD
duPrim IMUL = IMUL
duPrim INEG = INEG

-- Retype a source-language primitive operation as a target-language one
-- working on primal values. This is all the same because of 'niceprim'.
d1Prim : {σ τ : Typ Pr} -> Primop Pr σ τ -> Primop Du (D1τ σ) (D1τ τ)
d1Prim {σ} {τ} op =
  subst (\t -> Primop Du t (D1τ τ)) (sym (fst (niceprim op))) $
  subst (\t -> Primop Du (dut σ) t) (sym (snd (niceprim op))) $
    duPrim op


-------------------- EVALUATION --------------------------------------------------

-- A valuation / value environment: one value for each type in the typing
-- environment.
data Val (tag : PDTag) : Env tag -> Set where
  empty : Val tag []
  push : {Γ : Env tag} {τ : Typ tag} -> Rep τ -> Val tag Γ -> Val tag (τ :: Γ)

-- Project a value from a valuation.
valprj : ∀ {tag} {Γ : Env tag} {τ : Typ tag}
      -> (env : Val tag Γ) -> Idx Γ τ -> Rep τ
valprj (push x env) Z = x
valprj (push x env) (S i) = valprj env i

-- Map 'primal' over a valuation, lifting a valuation from the
-- non-differentiated world into a valuation in source-language world.
primalVal : {Γ : Env Pr} -> Val Pr Γ -> Val Du (D1Γ Γ)
primalVal empty = empty
primalVal {τ :: _} (push x env) = push (primal τ x) (primalVal env)

-- Given an inclusion of Γ' in Γ, and a valuation of Γ, build a valuation of
-- Γ'. This is used for evaluation of closures in 'eval' below.
```

```
buildValFromInj
  : ∀ {tag} {Γ Γ' : Env tag}
 -> ({ρ : Typ tag} -> Idx Γ' ρ -> Idx Γ ρ) -> Val tag Γ -> Val tag Γ'
buildValFromInj {Γ' = []} inj env = empty
buildValFromInj {Γ' = τ :: Γ'} inj env =
  push (valprj env (inj Z))
       (buildValFromInj (inj ∘ S) env)

-- The semantics of the term language. Aside from returning the evaluation
-- result, this also returns an integer recording the number of evaluation
-- steps taken during evaluation. This integer is used for complexity analysis.
eval : ∀ {tag} {Γ : Env tag} {τ : Typ tag}
    -> Val tag Γ -> Term tag Γ τ -> Rep τ × ℤ
eval env (var i) = valprj env i , one
eval env (let' rhs e) =
  let rhs' , crhs = eval env rhs
      e' , ce = eval (push rhs' env) e
  in e' , one + crhs + ce
eval env (lam Γ' inj e) =
  (\x -> eval (push x (buildValFromInj inj env)) e) , one + + (length Γ')
eval env (app e1 e2) =
  let f , cf = eval env e1
      x , cx = eval env e2
      y , cy = f x
  in y , one + cf + cx + cy
eval env (prim op e) =
  let e' , ce = eval env e
  in evalprim op e' , one + ce
eval env unit = tt , one
eval env (pair e1 e2) =
  let e1' , ce1 = eval env e1
      e2' , ce2 = eval env e2
  in (e1' , e2') , one + ce1 + ce2
eval env (fst' e) =
  let e' , ce = eval env e
  in fst e' , one + ce
eval env (snd' e) =
  let e' , ce = eval env e
  in snd e' , one + ce
eval env (inl e) =
  let e , ce = eval env e
  in inj₁ e , one + ce
eval env (inr e) =
  let e , ce = eval env e
  in inj₂ e , one + ce
eval env (case' e1 e2 e3) =
  let v , cv = eval env e1
  in case v of
       λ where (inj₁ x) -> let z , cz = eval (push x env) e2
                           in z , one + cv + cz
               (inj₂ y) -> let z , cz = eval (push y env) e3
                           in z , one + cv + cz
eval env (pureevm {Γ' = Γ'} e) =
  let e' , ce = eval env e
  in LACM.pure e' , one + ce
eval env (bindevm {Γ' = Γ'} e1 e2) =
  let e1' , ce1 = eval env e1
      e2' , ce2 = eval env e2
  in LACM.bind e1' e2' , one + ce1 + ce2
eval env (runevm {Γ' = Γ'} e1 e2) =
  let mf , ce1 = eval env e1
      denv , cdenv = eval env e2
```

```
      x , envctg , capp = LACM.run mf (LEtup-to-LEτ Γ' denv)
  in (x , LEτ-to-LEtup Γ' envctg) , one + ce1 + cdenv + capp
eval env (addevm {Γ' = Γ'} idx e) =
  let e' , ce = eval env e
  in LACM.add idx e' , one + ce
eval env (scopeevm e1 e2) =
  let e1' , ce1 = eval env e1
      e2' , ce2 = eval env e2
  in LACM.scope e1' e2' , one + ce1 + ce2
eval env lunit = tt , one
eval env (lpair e1 e2) =
  let e1' , ce1 = eval env e1
      e2' , ce2 = eval env e2
  in (just (e1' , e2')) , one + ce1 + ce2
eval env (lfst' {σ = σ} e) =
  let e' , ce = eval env e
  in case e' of
       λ where nothing -> let z , cz = zerov σ
                          in z , one + ce + cz
               (just (x , y)) -> x , one + ce
eval env (lsnd' {τ = τ} e) =
  let e' , ce = eval env e
  in case e' of
       λ where nothing -> let z , cz = zerov τ
                          in z , one + ce + cz
               (just (x , y)) -> y , one + ce
eval env lpairzero = nothing , one
eval env (linl e) =
  let e' , ce = eval env e
  in just (inj₁ e') , one + ce
eval env (linr e) =
  let e' , ce = eval env e
  in just (inj₂ e') , one + ce
eval env (lcastl {σ = σ} e) =
  let e' , ce = eval env e
  in case e' of
       λ where nothing ->
                  let z , cz = zerov σ
                  in z , one + ce + cz
               (just (inj₁ x)) -> x , one + ce
               (just (inj₂ _)) ->  -- NOTE: proper implementation would error
                  let z , cz = zerov σ
                  in z , one + ce + cz
eval env (lcastr {τ = τ} e) =
  let e' , ce = eval env e
  in case e' of
       λ where nothing ->
                  let z , cz = zerov τ
                  in z , one + ce + cz
               (just (inj₁ _)) ->  -- NOTE: proper implementation would error
                  let z , cz = zerov τ
                  in z , one + ce + cz
               (just (inj₂ y)) -> y , one + ce
eval env lsumzero = nothing , one

-- Project out the number of evaluation steps from 'eval'.
cost : ∀ {tag} {Γ : Env tag} {τ : Typ tag} -> Val tag Γ -> Term tag Γ τ -> ℤ
cost env e = snd (eval env e)


-------------------- CHAD ------------------------------------------------------
```

```
-- A term that produces the zero value of the given type.
zerot : {Γ : Env Du} -> (τ : Typ Pr) -> Term Du Γ (D2τ τ)
zerot Un = lunit
zerot Inte = lunit
zerot R = prim LZERO lunit
zerot (σ :* τ) = lpairzero
zerot (σ :+ τ) = lsumzero

-- The CHAD code transformation.
chad : {Γ : Env Pr} {τ : Typ Pr}
    -> Term Pr Γ τ
    -> Term Du (D1Γ Γ) (D1τ τ :* (D2τ τ :-> D2Γ Γ))
chad (var idx) =
  pair (var (convIdx D1τ idx))
       (lamwith [] (addevm (convIdx D2τ' idx) (var Z)))
chad (let' {σ = σ} e1 e2) =
  let' (chad e1) $
  let' (fst' (var Z)) $
  let' (sink (WCopy (WSkip WEnd)) (chad e2)) $
    pair (fst' (var Z))
         (lamwith (zero :: suc (suc zero) :: []) $
            bindevm
              (scopeevm (zerot σ) (app (snd' (var (S Z))) (var Z)))
              (lamwith (suc (suc zero) :: []) $
                 app (snd' (var (S Z))) (fst' (var Z))))
chad (prim op e) =
  let' (chad e) $
    pair (prim (d1Prim op) (fst' (var Z)))
         (lamwith (zero :: []) $
            app (snd' (var (S Z)))
                (dprim op (fst' (var (S Z))) (var Z)))
chad unit = pair unit (lamwith [] (pureevm unit))
chad (pair e1 e2) =
  let' (pair (chad e1) (chad e2)) $
    pair (pair (fst' (fst' (var Z)))
               (fst' (snd' (var Z))))
         (lamwith (zero :: []) $
            thenevm (app (snd' (fst' (var (S Z)))) (lfst' (var Z)))
                    (app (snd' (snd' (var (S Z)))) (lsnd' (var Z))))
chad (fst' {τ = τ} e) =
  let' (chad e) $
    pair (fst' (fst' (var Z)))
         (lamwith (zero :: []) $
            app (snd' (var (S Z)))
                (lpair (var Z) (zerot τ)))
chad (snd' {σ = σ} e) =
  let' (chad e) $
    pair (snd' (fst' (var Z)))
         (lamwith (zero :: []) $
            app (snd' (var (S Z)))
                (lpair (zerot σ) (var Z)))
chad (inl e) =
  let' (chad e) $
    pair (inl (fst' (var Z)))
         (lamwith (zero :: []) $
            app (snd' (var (S Z)))
                (lcastl (var Z)))
chad (inr e) =
  let' (chad e) $
    pair (inr (fst' (var Z)))
         (lamwith (zero :: []) $
            app (snd' (var (S Z)))
```

```
               (lcastr (var Z)))
chad (case' {σ = σ} {τ = τ} e1 e2 e3) =
  let' (chad e1) $
    case' (fst' (var Z))
      (let' (sink (WCopy (WSkip WEnd)) (chad e2)) $
         pair (fst' (var Z))
              (lamwith (zero ∷ suc (suc zero) ∷ []) $
                 bindevm
                   (scopeevm (zerot σ) (app (snd' (var (S Z))) (var Z)))
                   (lamwith (suc (suc zero) ∷ []) $
                      app (snd' (var (S Z))) (linl (fst' (var Z))))))
      (let' (sink (WCopy (WSkip WEnd)) (chad e3)) $
         pair (fst' (var Z))
              (lamwith (zero ∷ suc (suc zero) ∷ []) $
                 bindevm
                   (scopeevm (zerot τ) (app (snd' (var (S Z))) (var Z)))
                   (lamwith (suc (suc zero) ∷ []) $
                      app (snd' (var (S Z))) (linr (fst' (var Z))))))


-------------------- THE COMPLEXITY THEOREMS ----------------------------------

-- The potential function, here using c_φ = 1 because this suffices due to our
-- costing of plusv.
φ : (τ : LTyp) -> LinRep τ -> ℤ
φ LUn tt = one
φ LR _ = one
φ (σ :*! τ) nothing = one
φ (σ :*! τ) (just (x , y)) = one + φ σ x + φ τ y
φ (σ :+! τ) nothing = one
φ (σ :+! τ) (just (inj₁ x)) = one + φ σ x
φ (σ :+! τ) (just (inj₂ y)) = one + φ τ y

-- The potential function mapped over a list of linear types.
φ' : (Γ : LEnv) -> LEtup Γ -> ℤ
φ' [] tt = + 0
φ' (τ ∷ Γ) (x , env) = φ τ x + φ' Γ env

-- The statement of the complexity theorem including potential. A value of this
-- type (i.e. a proof of the theorem) is given in `chad-cost.agda`.
TH1-STATEMENT : Set
TH1-STATEMENT =
  {Γ : Env Pr} {τ : Typ Pr}
  -> (env : Val Pr Γ)
  -> (ctg : Rep (D2τ τ))
  -> (denvin : LEtup (map D2τ' Γ))
  -> (t : Term Pr Γ τ)
  -> let (_primal , bp) , crun = eval (primalVal env) (chad t)
         envf , ccall = bp ctg
         tt , denvout , cmonad = LACM.run envf denvin
     in
     crun
      + ccall
      + cmonad
      - φ (D2τ' τ) ctg
      - φ' (map D2τ' Γ) denvin
      + φ' (map D2τ' Γ) denvout
      - + length Γ
      ≤ + 34 * cost env t

-- In th2 we bound φ by the size of the incoming cotangent. This measures the
-- size of a cotangent value.
```

```
size : (τ : LTyp) -> LinRep τ -> ℕ
size LUn .tt = 1
size LR _ = 1
size (σ :*! τ) nothing = 1
size (σ :*! τ) (just (x , y)) = 1 +ℕ size σ x +ℕ size τ y
size (σ :+! τ) nothing = 1
size (σ :+! τ) (just (inj₁ x)) = 1 +ℕ size σ x
size (σ :+! τ) (just (inj₂ y)) = 1 +ℕ size τ y

-- In th2 we initialise the environment derivative accumulator to zero, because
-- that is how CHAD will be used in practice. This term creates a zero
-- environment derivative.
zero-env-term : {Γ' : Env Du} -> (Γ : Env Pr) -> Term Du Γ' (D2Γtup Γ)
zero-env-term [] = unit
zero-env-term (τ :: Γ) = pair (zerot τ) (zero-env-term Γ)

-- The statement of the corollary that bounds φ to not mention potential any
-- more. A value of this type (i.e. a proof of the theorem) is given in
-- `chad-cost.agda`.
TH2-STATEMENT : Set
TH2-STATEMENT =
  {Γ : Env Pr} {τ : Typ Pr}
  -> (env : Val Pr Γ)
  -> (ctg : Rep (D2τ τ))
  -> (t : Term Pr Γ τ)
  -> cost (push ctg (primalVal env))
          (runevm (app (snd' (sink (WSkip WEnd) (chad t)))
                       (var Z))
                  (zero-env-term Γ))
     ≤ + 5
        + + 34 * cost env t
        + + size (D2τ' τ) ctg
        + + 4 * + length Γ
```

6.10.5 EVM implementation in GHC Haskell

The implementation spans two modules: `EVM_IO`, the actual monad implementation, and `ParMonoid`, a type class (with instances) for types supporting parallel accumulation.

```
{-# LANGUAGE DataKinds, DerivingStrategies, GADTs, GeneralizedNewtypeDeriving,
    ImportQualifiedPost, KindSignatures, RoleAnnotations, ScopedTypeVariables,
    TypeApplications, TypeOperators #-}

{-|
Environment vector monad, built using 'IO'. This implementation supports
parallelism. Dependencies are the "transformers" and "vector" packages, and
ParMonoid additionally depends on "stm".

Internally this code uses 'unsafeCoerce' quite heavily, but the exposed API is
type-safe.
-}
module EVM_IO (
  -- * EVM
  EVM,
  run,
  one,
  scope,
  parsequence,

  -- * Supporting types
  HList(..),
  hlength,
  Idx(..),
) where

import Control.Monad              (forM_)
import Control.Monad.Trans.Class  (lift)
import Control.Monad.Trans.Reader (ReaderT, runReaderT, local, ask)
import Data.Kind                  (Type)
import Data.Vector                qualified as V
import Data.Vector.Mutable        qualified as MV
import Data.IORef                 (IORef, newIORef, readIORef, writeIORef)
import GHC.Exts                   (Any)
import Unsafe.Coerce              qualified as Unsafe (unsafeCoerce)

import ParMonoid

-- | The environment vector monad.
newtype EVM (ts :: [Type]) a =
    -- The 'Int' is the length of 'ts'.
    -- The IORef contains the data: head of type-level list = end of the array.
    --   It is wrapped in an IORef so that we can reallocate it non-lexically.
    --   The 'Any' is the corresponding 't' from 'ts'.
    EVM (ReaderT (Int, IORef (MV.IOVector Any)) IO a)
```

```
  deriving newtype (Functor, Applicative, Monad)

-- Raise the 'ts' argument from 'phantom' to 'representational' to prevent
-- coercion to unrelated types, which would invalidate the uses of
-- 'unsafeCoerce' in this file.
type role EVM representational _

-- | A heterogeneous list.
data HList (ts :: [Type]) where
  HNil :: HList '[]
  HCons :: t -> HList ts -> HList (t ': ts)

hlength :: HList ts -> Int
hlength HNil = 0
hlength (HCons _ l) = 1 + hlength l

-- | An index into a type-level list.
data Idx (ts :: [Type]) t where
  Z :: Idx (t ': ts) t
  S :: Idx ts t -> Idx (u ': ts) t


-- | Handler of the 'EVM' monad. The integer is the initial capacity of the
-- environment array, if it is larger than the input 'HList'.
run :: Int -> EVM ts a -> HList ts -> IO (a, HList ts)
run initcap m initvec = do
  -- allocate the array
  let initlen = hlength initvec
  arr <- MV.replicate (max initcap initlen) dummy
  -- fill in the initial values; need to reverse because the head of the list
  -- should be at the end of the array
  forM_ (zip (reverse (anonymiseHList initvec)) [0..]) $ \(x, i) ->
    MV.write arr i x
  -- run the computation
  ref <- newIORef arr
  result <- let EVM f = m in runReaderT f (initlen, ref)
  -- read the final values; again, reversed
  outvec <- deanonymiseHList initvec <$>
              traverse (MV.read arr) [initlen-1, initlen-2 .. 0]
  -- return
  return (result, outvec)
  where
    anonymiseHList :: HList ts -> [Any]
    anonymiseHList HNil = []
    anonymiseHList (HCons x xs) = unsafeToAny x : anonymiseHList xs

    deanonymiseHList :: HList ts -> [Any] -> HList ts
    deanonymiseHList HNil _ = HNil
    deanonymiseHList (HCons _ model) (x : xs) =
      HCons (unsafeFromAny x) (deanonymiseHList model xs)
    deanonymiseHList (HCons _ _) [] = error "Will not happen"
```

```haskell
-- | Accumulate a value into the environment vector.
one :: forall t ts. ParMonoid t => t -> Idx ts t -> EVM ts ()
one x idx = EVM $ do
  (veclen, arrref) <- ask
  -- compute the index in the array
  let idx' = veclen - 1 - idx2int idx
  -- read the current value from the array
  arr <- lift $ readIORef arrref
  var <- unsafeFromAny @(VarRepr t) <$> MV.read arr idx'
  -- update it with the combined result
  lift $ pmappend var x
  where
    idx2int :: Idx ts' i -> Int
    idx2int Z = 0
    idx2int (S i) = 1 + idx2int i

-- | Enter a scope with an additional cell in the accumulator array. The given
-- value is the initial value of the new cell.
scope :: forall t ts a. ParMonoid t => t -> EVM (t ': ts) a -> EVM ts (a, t)
scope zero m = EVM $ do
  (veclen, arrref) <- ask
  -- if we're at capacity, reallocate and update 'arrref'
  arr <- lift $ readIORef arrref
  let capacity = MV.length arr
  arr2 <- if veclen == capacity
          then do arr2 <- lift $ resize (2 * capacity) dummy arr
                  lift $ writeIORef arrref arr2
                  return arr2
          else return arr
  -- now that there's space, write the zero value in the next slot
  var <- lift $ pmakevar zero
  MV.write arr2 veclen (unsafeToAny @(VarRepr t) var)
  -- run the computation (note that 'arrref' was already updated above)
  result <- let EVM f = m in local (\_ -> (veclen + 1, arrref)) f
  -- read the result value
  outval <- lift $ pfinalise var
  -- return
  return (result, outval)

-- | Evaluate a vector of computations in parallel.
--
-- The given function is a parallelism strategy; it should evaluate the vector
-- of IO operations in parallel in IO somehow.
--
-- Technically this function allows you to evaluate arbitrary IO actions within
-- EVM by exploiting the flexible typing of the parallelism strategy. This is,
-- however, not the intent because we're unsure whether doing so is
-- semantically sound.
parsequence :: (V.Vector (IO a) -> IO (V.Vector a))
            -> V.Vector (EVM ts a) -> EVM ts (V.Vector a)
```

```
parsequence parallel vm = EVM $ do
  (veclen, arrref) <- ask  -- we're unlifting through ReaderT to IO here
  lift $ parallel (V.map (\(EVM f) -> runReaderT f (veclen, arrref)) vm)

-- | Resize an 'IOVector' to the given bounds, filling new cells with the given
-- initial value.
resize :: Int -> e -> MV.IOVector e -> IO (MV.IOVector e)
resize newlen initval arr = do
  let len = MV.length arr
  arr2 <- MV.generateM newlen $ \i ->
            if i < len then MV.read arr i
                       else return initval
  return arr2

dummy :: Any
dummy = unsafeToAny ()

unsafeToAny :: a -> Any
unsafeToAny = Unsafe.unsafeCoerce

unsafeFromAny :: Any -> a
unsafeFromAny = Unsafe.unsafeCoerce
```

```haskell
{-# LANGUAGE LambdaCase, TypeFamilies #-}
module ParMonoid where

import Control.Concurrent.STM
import Control.Monad          (when)

class ParMonoid a where
  type VarRepr a
  -- | Allocate a mutable cell.
  pmakevar :: a -> IO (VarRepr a)
  -- | Read the final value out of a mutable cell; this is NOT thread-safe!
  pfinalise :: VarRepr a -> IO a
  -- | Add a value into a mutable cell. This is thread-safe.
  pmappend :: VarRepr a -> a -> IO ()

instance ParMonoid () where
  type VarRepr () = ()
  pmakevar _ = return ()
  pfinalise _ = return ()
  pmappend _ _ = return ()

instance ParMonoid Float where
  type VarRepr Float = TVar Float
  pmakevar x = newTVarIO x
  pfinalise p = atomically $ readTVar p
  pmappend p x = atomically $ modifyTVar' p (+ x)

instance (ParMonoid a, ParMonoid b) => ParMonoid (a, b) where
  type VarRepr (a, b) = (VarRepr a, VarRepr b)
  pmakevar (x, y) = (,) <$> pmakevar x <*> pmakevar y
  pfinalise (p1, p2) = (,) <$> pfinalise p1 <*> pfinalise p2
  pmappend (p1, p2) (x, y) = pmappend p1 x >> pmappend p2 y

pmappendSparse
  :: (a -> IO va) -> (va -> a -> IO ()) -> TMVar va -> Maybe a -> IO ()
pmappendSparse _ _ _ Nothing = return ()
pmappendSparse makevar append p (Just x) =
  atomically (tryReadTMVar p) >>= \case
    Nothing -> do
      var <- makevar x
      success <- atomically $ tryPutTMVar p var
      when (not success) $ do
        var' <- atomically $ readTMVar p
        append var' x
    Just var -> append var x

instance ParMonoid a => ParMonoid (Maybe a) where
  type VarRepr (Maybe a) = TMVar (VarRepr a)
  pmakevar Nothing = newEmptyTMVarIO
  pmakevar (Just x) = pmakevar x >>= newTMVarIO
  pfinalise p = atomically (tryReadTMVar p) >>= \case
```

```
                Nothing -> return Nothing
                Just var -> Just <$> pfinalise var
  pmappend = pmappendSparse pmakevar pmappend

newtype LEither a b = LEither (Maybe (Either a b))
  deriving (Show)

instance (ParMonoid a, ParMonoid b) => ParMonoid (LEither a b) where
  type VarRepr (LEither a b) = TMVar (Either (VarRepr a) (VarRepr b))
  pmakevar (LEither Nothing) = newEmptyTMVarIO
  pmakevar (LEither (Just (Left x))) = pmakevar x >>= newTMVarIO . Left
  pmakevar (LEither (Just (Right x))) = pmakevar x >>= newTMVarIO . Right
  pfinalise p = atomically (tryReadTMVar p) >>= \case
                Nothing -> return (LEither Nothing)
                Just (Left var) -> LEither . Just . Left <$> pfinalise var
                Just (Right var) -> LEither . Just . Right <$> pfinalise var
  pmappend p (LEither mx) = pmappendSparse makeContents addContents p mx
    where
      makeContents (Left x) = Left <$> pmakevar x
      makeContents (Right x) = Right <$> pmakevar x

      addContents (Left var) (Left y) = pmappend var y
      addContents (Right var) (Right y) = pmappend var y
      addContents (Left _) (Right _) = error "ParMonoid: Left + Right"
      addContents (Right _) (Left _) = error "ParMonoid: Right + Left"
```

7 Fast CHAD: Making It Practical

The main research work of this thesis started in Chapter 3, where we analysed the complexity of dual-numbers reverse AD and improved it to satisfy the optimality requirement: a gradient is computed with a constant-factor overhead in computation steps relative to the original. We then proceeded in Chapter 4 by noting that asymptotic efficiency does not imply practical efficiency; to address this problem, we spent time on an algorithm derived from dual-numbers reverse AD that has more favourable performance characteristics by means of a rewrite system that splits array combinators into bulk array operations.

Naive CHAD required more than a simple introductory section, so we spent Chapter 5 on the details of the algorithm and why it works as it does. For Efficient CHAD (Chapter 6), we then proceeded as in Chapter 3 by identifying the complexity issues and fixing them. In this case, that included a formalised proof that we succeeded, at least on the first-order fragment of the source language. Altogether, this means that in Chapter 7 it is now time to declare the algorithm of Chapter 6 too slow for practical use and consider ways to improve the situation again.

Like in Chapter 4, the optimisations that we present in this chapter are not effective on the full, higher-order source language of naive CHAD, and furthermore the optimised algorithm is more complex than what we obtained in Chapter 6 when thinking only about complexity. Unlike the bulk dual-numbers algorithm in Chapter 4, however, the algorithm remains applicable to the full source language: it may just produce inefficient (or even slow) output code when applied to programs outside the favourable fragment, which we specify precisely in Section 7.1. Furthermore, the algorithm subjectively retains "more" compositionality than we managed in Chapter 4, although we must leave it to the reader to be the final judge on this point.

In one final way, this chapter is simultaneously like and unlike Chapter 4: we

This chapter describes in-progress research by the author of the thesis.

describe a work-in-progress, and the list of future work at the end of the chapter is long. On the other hand, at least some of those weaknesses seem solvable — a situation that was much more nuanced in Chapter 4.

Summary of contributions. We incrementally improve the CHAD algorithm in a number of steps; some of these are orthogonal.

- In Section 6.6, we used $\mathcal{D}[\text{Array } \tau]_2 = \text{Bag } (\mathbb{Z} \times \mathcal{D}[\tau]_2)$ to make array cotangents sparse and preserve our complexity guarantees, but this definition is untenable when considering practical performance, especially on parallel hardware. Thus, we start in Section 7.2 by reverting to simply $\mathcal{D}[\text{Array } \tau]_2 = \mathbf{1} \sqcup \text{Array } \mathcal{D}[\tau]_2$. This breaks our complexity results, but on our "favourable language" — specified in Section 7.1 and based on Accelerate [Chakravarty et al. 2011] — we can avoid complexity issues in almost all cases.

- The Efficient CHAD algorithm of Chapter 6 created backpropagator closures for every source language term. This is wasteful and in fact unnecessary; in Section 7.3, we eliminate backpropagator lambdas by inlining them. This inlining is justified by ensuring that backpropagator lambdas are invoked in exactly one place in the program, meaning that inlining does not increase code size. This single-invocation property is already true for straight-line code, but some work must be done to make it true also for control-flow constructs. As a result, we avoid creating a closure at all for first-order source terms.

- While a closure (implicitly) selectively stores only some of the bindings in the environment, the reformulated code transformation now stores *all* let-bindings shared between the primal and the dual, regardless of whether the backpropagator needs them. To reduce memory traffic and peak memory consumption, Section 7.4 reintroduces the ability to store only a subset of these bindings, but now explicitly in the code transformation.

We have a prototype implementation of the algorithm derived in this chapter that generates (somewhat naive) C code from the compiled result. This implementation is briefly discussed in Section 7.5.

As part of this implementation, more improvements to CHAD were made than are listed above, and some of these are significant. The most interesting features are sketched in Section 7.5.1, with a more thorough discussion deferred to future work. Summarised, these features are:

- Section 7.5.1.1: Selectively reverting, for a subset of the environment, to the naive CHAD behaviour of simply returning cotangents from the backpropagator, thus avoiding effectful accumulation. Judicious use of this

functionality significantly simplifies the derivative code, with attendant performance improvements stemming from the increased transparency of the computation to the compiler.

- Section 7.5.1.2: Mixed static-dynamic sparsity of cotangents. The dynamic cotangent sparsity of Chapter 6 ($\mathcal{D}[\sigma \times \tau]_2 = \mathbf{1} \sqcup (\mathcal{D}[\sigma]_2 \times \mathcal{D}[\tau]_2)$) is overkill in many practical situations, as we often know already statically that a certain cotangent will always be absent (inl $\langle\rangle$) or always present (inr _). With this knowledge, we can elide the majority of such dynamic sparsity wrappers, essentially moving many of the sparsity decisions from runtime to compile time. This technique is particularly effective after the previous step, as many statically-known zeros were introduced by that change. This kind of "optimised sparsity" for algebraic data types may be of independent interest outside of CHAD.

- Additionally, we implement some optimisations and features that are practically useful but do not change the algorithm itself; described are: a fine-granularity dead-code elimination pass that is particularly helpful for CHAD-generated code (Section 7.5.1.3), a trick to "shortcut" the creation of redundant zeros for accumulation using a compile-time value identity analysis (Section 7.5.1.4) and an extremely simple and obviously correct, yet already useful, implementation of checkpointing (Section 7.5.1.5).

7.1 Generality, performance, and the source language

The source language we have worked on so far in Chapters 5 and 6 is rather ambitious, including support for unrestricted lambda abstraction and application, nested arrays and second-order array operations on them, as well as coproduct types. Implementations of reverse AD that attain good performance in practice are, to our knowledge, limited to define-then-run methods (Section 2.2.10) that operate on more restricted languages, and taping methods (Section 2.2.8) that differentiate at runtime. The algorithm we derive in this chapter is no different and falls in the first category, with support for second-order array combinators but without (efficient) support for nested arrays or higher-order functions. Its novelty over the state of the art in performant reverse AD lies in its strong link with theory.

The presentation of the algorithm in this chapter follows our implementation (Section 7.5) and is explained with respect to two languages:

1. The language from Chapter 6 minus function types; that is: Chapter 5 minus lambda abstraction and application, but including arrays (Section 6.6). On

$$
\begin{array}{rll}
\rho & ::= \mathbb{R} \mid \mathbb{Z} \mid \mathbf{1} \mid \rho_1 \times \rho_2 \mid \rho_1 \sqcup \rho_2 & \text{(element types)} \\
\tau, \sigma & ::= \text{Array } \rho \mid \mathbf{1} \mid \sigma \times \tau \mid \sigma \sqcup \tau & \text{(array types)} \\
t, s & ::= \langle\rangle \mid \langle s, t \rangle \mid \text{fst } t \mid \text{snd } t \mid \text{inl } t \mid \text{inr } t & \text{(array programs)} \\
& \quad \mid \textbf{case } s \textbf{ of } \{ \text{ inl } a_1 \to t_1 \mid \text{inr } a_2 \to t_2 \ \} & \\
& \quad \mid \textbf{let } a = s \textbf{ in } t \mid a & \text{(let-binding and variables)} \\
& \quad \mid \text{build } e_1 \ (i.\ e_2) \mid \text{fold } (x.\ e)\ t & \\
e & ::= \langle\rangle \mid \langle e_1, e_2 \rangle \mid \text{fst } e \mid \text{snd } e \mid \text{inl } e \mid \text{inr } e & \text{(expressions)} \\
& \quad \mid \textbf{case } e \textbf{ of } \{ \text{ inl } x_1 \to e_1 \mid \text{inr } x_2 \to e_2 \ \} & \\
& \quad \mid \textbf{let } x = e_1 \textbf{ in } e_2 \mid x & \text{(let-binding and variables)} \\
& \quad \mid r \mid n \mid \text{op}_{\mathbb{R}}(e_1, \ldots, e_n) \mid \text{op}_{\mathbb{Z}}(e_1, \ldots, e_n) & \\
& \quad \mid a \,!\, e \mid \text{length } a & \text{(array vars. in exprs.)}
\end{array}
$$

Figure 7.1: The grammar of Idealised Accelerate.

this language, the transformation should be correct (not formally proved), but it does not necessarily have the right complexity. This limitation is a consequence of practical efficiency trade-offs made in favour of the second language:

2. A smaller language based on Accelerate [Chakravarty et al. 2011][1] called *Idealised Accelerate* (occasionally *IA*),[2] on which we endeavour to be practically efficient and close to the optimal complexity. This is a second-order array language in the sense of Section 2.1.1; its grammar is shown in Fig. 7.1.

Idealised Accelerate differs from the language in Chapter 6 by disallowing nested arrays and enforcing a strict separation between *array programs* that compute with arrays (τ) and *expressions* that compute with elements (ρ). This design is copied from Accelerate. In particular, this means that array variables a always have a type from the production τ and expression variables x always have a type from the production ρ. Expressions occur inside array programs, but may refer to the enclosing array program only by referring to array *variables* in indexing (!) and when querying the length of an array. This syntactic variable restriction is emphasised in the grammar by marking these locations in red. The internal language of the Accelerate compiler shares this variable restriction, and it will be important for our complexity reasoning, especially in Section 7.2.[3]

[1] Our target language, however, is larger than Accelerate.

[2] The master thesis of the author [Smeding 2021] also presented a language by this name, but it was actually rather close to the Accelerate compiler's internal representation and not very idealised.

[3] It is possible that this restriction can be loosened significantly or even eliminated completely; see future work in Section 7.6.

The most important omissions compared to Accelerate are multidimensional arrays, various second-order array operations and additional primitive numeric types. However, all of these could be supported using simple extensions to the algorithms presented in this chapter without interesting new problems for differentiation; we skip them so that we can focus on the concepts and not get lost in details. Ingenuity may be required for the array operations, but no more than for any other reverse AD algorithm; existing derivatives of such operations found in the literature (e.g. [Paszke et al. 2021b; Schenck et al. 2022]) should be portable without much trouble.

On the other hand, IA is more expressive than Accelerate in one way: we fully support coproducts, whereas the Accelerate compiler has only products in its internal language.[4]

Compared to the languages used before in Chapters 5 and 6, the most important omissions are lambda functions and nested arrays, as without nested arrays, the variable restriction in indexing is not a major loss in expressivity. Lambda abstraction is excluded not because there is anything wrong with the rules per se (after fixing the complexity issues using closure conversion, Section 6.8), but because higher-order code obscures the provenance of values, including arrays (which is detrimental to the analysis in Section 7.2.2) and because lambda functions complicate the propagation of static sparsity (see Section 7.5.1.2), which would become non-local. More difficulties might appear if these are solved, but we defer this to future work. Finally, the restriction to non-nested arrays is helpful for efficiently handling array indexing, but we conjecture that it is possible to do away with this restriction (Section 7.6).

7.2 Dense array cotangents

In Section 6.6, we gave the following definition for the type of cotangents to an array:

$$\mathcal{D}[\text{Array } \tau]_2 = \text{Bag } (\mathbb{Z} \times \mathcal{D}[\tau]_2)$$
$$\textbf{data } \text{Bag } \tau = \text{BEmpty} \mid \text{BOne } \tau \mid \text{BPlus } (\text{Bag } \tau)\ (\text{Bag } \tau)$$

This definition sufficed for optimal asymptotic complexity, but as already diagnosed in Section 6.7, it is unsuitable for practical efficiency. We briefly discussed mitigations in Section 6.7.1, including adding an additional constructor BArray (Array τ) to Bag, but in practice, any possibility of reverting to a big tree

[4]The Accelerate surface language supports coproducts, but n-ary coproducts $\tau_1 \sqcup \cdots \sqcup \tau_n$ are desugared to $\texttt{Int8} \times \tau_1 \times \cdots \times \tau_n$ before compilation proper.

of BOne and BPlus is unacceptable, and indeed hard to implement in the first place on wide-vector platforms like GPUs.[5]

In this chapter, we "bite the bullet" and revert to the straightforward definition:

$$\mathcal{D}[\text{Array } \tau]_2 = \mathbf{1} \sqcup \text{Array } \mathcal{D}[\tau]_2 \tag{7.1}$$

analogous to the rule for products from Chapter 6 (Section 6.3.1):

$$\mathcal{D}[\sigma \times \tau]_2 = \mathbf{1} \sqcup (\mathcal{D}[\sigma]_2 \times \mathcal{D}[\tau]_2)$$

Equation (7.1) has the effect of simplifying, and at the same time improving the performance of, $\overline{\mathcal{D}}_\Gamma[\text{build}]$ and other array operations, as no more traversal of a Bag structure is needed. The significant downside, however, is that array indexing reverts to its naive one-hot derivative:

$$\begin{aligned}\mathcal{D}_\Gamma[s\,!\,t] = \mathbf{let}\ &\langle x_1, x_2\rangle = \mathcal{D}_\Gamma[s]\\ &\langle i, \_\rangle = \mathcal{D}_\Gamma[t]\\ \mathbf{in}\ &\langle x_1\,!\,i, \lambda d.\ x_2\ (\text{inr } (\text{build } (\text{length } x_1)\ (j.\ \mathbf{if}\ j = i\ \mathbf{then}\ d\ \mathbf{else}\ \underline{0})))\rangle\end{aligned}$$

It seems that the question of what to do with array indexing confronts every reverse AD algorithm that tries be fast and (purely) functional at the same time. There is a good explanation for this: as reverse AD reverses the data flow graph of a program (execution),[6] reads in the source program turn into writes (more precisely, accumulation using (+)) in the reverse pass of the derivative program. If we track individual cotangents for all values that can be read individually in the source language, as we did in Chapter 3, we naturally obtain sufficiently granular writes; the downside of this approach is bad performance on (bulk) array operations (see Section 4.3.1). Once we try to solve this administration overhead by lifting the granularity of array cotangents from elements to the entire array, however, tension materialises with the reverse derivative of read operations on *elements* of an array — that is, array indexing (see Section 4.3.2).

We can extract a lesson from this:

Observation. If the source language is allowed to read from a subobject in constant time, then the target language must be able to write to, or at least add to, such a subobject in constant time.[7]

The solution of Chapter 6 using Bag works around the problem of not having cotangent accumulation at the granularity of array elements by paying a

[5] A tree structure hinders flat, regular parallelism, which is necessary to perform well on GPUs, even if one manages to avoid pointer indirections in the tree representation.

[6] Recall Section 2.2.6.

[7] This is not a problem with pairs as constant-time zeros allow creating a one-hot pair in $O(1)$.

slowness cost, similar to the inefficiency in Chapter 3. Practical, fast reverse AD implementations use mutation here, and we do the same in this chapter.

Fortunately, Chapter 6 has given us most of the infrastructure for this: extending the **one** method of our local accumulation monad **EVM** to allow accumulation into subobjects (Section 7.2.1) is sufficient to make the derivative of array indexing constant-time on Idealised Accelerate — as long as an array to accumulate into is already present.[8] The first subobject accumulation into a certain array cotangent may need to allocate this array if it is still sparse; partial mitigations are discussed in Section 7.2.2.

7.2.1 Subobject accumulation

To imperative programmers, the read–write duality on array elements is utterly natural as it is, in fact, present in the syntax of many programming languages: the read statement "$x = a[i]$" can be turned into a write "$a[i] = x$" by merely flipping the operands of the assignment operator. The special case of writing that we need — accumulation — is written simply "$a[i] \mathrel{+}= x$". In fact, this syntax commonly goes further: "$struc.abc[i].def$" is composite syntax for reading the *def* field of the i'th index of the array in the *abc* field of *struc*. Our source language expresses the read side of this duality (although our product types, with just fst and snd, are a bit spartan), so it remains to design a target language primitive that implements the write side of the duality.

Our mutable accumulation method, the **one** method of **EVM**, currently allows accumulation to the entire cotangent for an environment entry, not to a subobject of such an entry. To support subobject accumulation, we replace **one** with an indexed family of operations, indexed by the *location* in the accumulator where the argument should be added. Equivalently, we add an argument to **one** describing such a location, with the restriction that this argument be fully statically known, apart from the actual array indices.

We describe these locations using accumulator projections, defined in Fig. 7.2.[9] These projections "point to" a smaller monoid value inside of a larger one. The 'acprj' grammar describes the static part of a location, including the field in product types into which to descend but excluding the index in an array into which to descend. Specifically, $\overleftarrow{\times}$ and $\overrightarrow{\times}$ point to the first, respectively the second

[8] Our implementation ensures this by initialising **scope** with a dense zero; see Sections 7.5.1.2 and 7.5.1.4.

[9] This grammar of projections describes more than just array indexing; for IA, acprj ::= $*$ | $!*$ would suffice. With some simple compiler optimisations (see 'Generalisation' on page 312), the more general version allows efficiently handling a slightly larger fragment of the source language without introducing much more machinery. Efficiently handling even more of the source language is harder, and future work. In particular, adding coproduct projections does not help, as the **onehot** usages in $\mathcal{D}_\Gamma[\mathbf{case}]$ cannot fuse with an **accum** (introduced later in this section).

$$
\begin{aligned}
&\text{acprj} ::= * \mid \overleftarrow{\times}\,\text{acprj} \mid \overrightarrow{\times}\,\text{acprj} \mid !\,\text{acprj}\\
&\begin{aligned}
&\text{APty} : \text{acprj} \to \text{Type} \to \text{Type}\\
&\text{APty}\ *\qquad \tau \qquad\qquad\quad = \tau\\
&\text{APty}\ (\overleftarrow{\times}\,p)\ (\mathbf{1} \sqcup (\sigma \times \tau)) \quad = \text{APty}\ p\ \sigma\\
&\text{APty}\ (\overrightarrow{\times}\,p)\ (\mathbf{1} \sqcup (\sigma \times \tau)) \quad = \text{APty}\ p\ \tau\\
&\text{APty}\ (!\,p)\ \ (\mathbf{1} \sqcup (\text{Array}\ \tau)) = \text{APty}\ p\ \tau
\end{aligned}\\
&\begin{aligned}
&\text{APix} : \text{acprj} \to \text{Type} \to \text{Type}\\
&\text{APix}\ *\qquad \tau \qquad\qquad\quad = \mathbf{1}\\
&\text{APix}\ (\overleftarrow{\times}\,p)\ (\mathbf{1} \sqcup (\sigma \times \tau)) \quad = \text{APix}\ p\ \sigma\\
&\text{APix}\ (\overrightarrow{\times}\,p)\ (\mathbf{1} \sqcup (\sigma \times \tau)) \quad = \text{APix}\ p\ \tau\\
&\text{APix}\ (!\,p)\ \ (\mathbf{1} \sqcup (\text{Array}\ \tau)) = \underbrace{\mathbb{Z}}_{\text{(index)}} \times \underbrace{\mathbb{Z}}_{\text{(length)}} \times \text{APix}\ p\ \tau
\end{aligned}
\end{aligned}
$$

Figure 7.2: Accumulator projections. The two integers in $\text{APix}\ (!\,p)\ (\mathbf{1} \sqcup (\text{Array}\ \tau))$ respectively record the index at which to accumulate, and the length of the array being accumulated into.

field of a sparse product ($\sigma \underline{\times} \tau$) as introduced in Section 6.3.1; ! points to a (runtime-determined) array element; and $*$ terminates the projection path.

There are two supporting dependent types. 'APty' computes the inner type that the projection points to, so that, for example:

$$\text{APty}\ (\overleftarrow{\times}\,!\,*)\ (\mathbf{1} \sqcup ((\mathbf{1} \sqcup \text{Array}\ \sigma) \times \tau)) = \sigma$$

Note that we descend into the full, sparse version of the type (with an adjoined $\mathbf{1}$) and not plain $\sigma \times \tau$ or $\text{Array}\ \tau$ as we want each step in an accumulator projection to be a monoid. While $\sigma \times \tau$ is a fine monoid if σ and τ are, treating it separately forces us to additionally separately treat the $\mathbf{1} \sqcup$ that wraps it, which introduces ambiguity with the monoid instance of coproducts.

'APix' specifies what additional runtime information is required to fully determine where to accumulate. As all location information apart from array indices is already contained in the statically-known acprj, APix only produces a type different from $\mathbf{1}$ if the projection path indexes into an array (i.e. contains a ! component). In fact, APix actually records not only the index at which to accumulate but also the length of the array being indexed into. This length is necessary because accumulation may need to unsparsify a $\mathcal{D}[\![\text{Array}\ \tau]\!]_2 = \mathbf{1} \sqcup \text{Array}\ \mathcal{D}[\![\tau]\!]_2$ by allocating an array of the right length.

Using these new types, we can define our subobject accumulation primitive, which, as stated, is a (family of) method(s) of **EVM**. We switch from '**one**' to the name '**accum**' to better reflect its operational behaviour. Its type is as follows:

$$
\begin{aligned}
&\mathbf{accum}_{(x:\tau\in\Gamma),p} : \text{APix}\ p\ \tau \to \text{APty}\ p\ \tau \to \mathbf{EVM}\ \Gamma\ \mathbf{1}\\
&\quad \mathbf{one}_{\ x:\tau\in\Gamma} \quad : \qquad\qquad\qquad\qquad \tau \to \mathbf{EVM}\ \Gamma\ \mathbf{1} \qquad \text{(from Chapter 6)}
\end{aligned}
$$

where p is an 'acprj' suitable for τ (i.e. such that APty $p\ \tau$ reduces). These two primitives are related by the equality:

$$\mathbf{accum}_{(x:\tau\in\Gamma),*}\ \langle\rangle\ x = \mathbf{one}_{x:\tau\in\Gamma}\ x$$

i.e. accumulating with the trivial projection, and hence trivial additional runtime information (as APix $*\ \tau = \mathbf{1}$), is equivalent to **one**.

The reader may wonder why we choose this particular split of location information for **accum** between the statically-known part (the p subscript, i.e. an instance of the 'acprj' grammar) and the runtime part (the APix $p\ \tau$ argument). The answer is that this split reflects the design of our source language: for a projection from a pair, we know statically which component it selects (by it being a 'fst' or a 'snd' term), whereas for array indexing, the precise index is a runtime value. As a result, the construction of the reverse derivative term has the direction of a pair projection available statically, but not (necessarily) the index into an array. As the nature of (heterogeneous) product types and (homogeneous) arrays in general strongly suggests this language design, the particular split seen here likely generalises to other languages.

Note that we want **accum**'s parametrisation to be as static as possible: the better we know the projection path statically, the less runtime branching is necessary in the implementation of **accum**. Since moving also array indices into **accum**'s static argument would make it useless for the derivative of array indexing, the optimum seems to be the split given here.

Example. Consider the following use of **one**:

$$\mathbf{one}_{x:\mathbf{1}\sqcup\mathrm{Array}\,(\mathbf{1}\sqcup(\mathbb{R}\times\mathbf{1}))\in\Gamma}\ (\mathrm{inr}\ (\mathrm{build}\ n\ (i.\ \mathbf{if}\ i = 4\ \mathbf{then}\ \mathrm{inr}\ \langle d, 0\rangle\ \mathbf{else}\ \mathrm{inl}\ \langle\rangle)))$$

Note that $\mathbf{1}\sqcup\mathrm{Array}\,(\mathbf{1}\sqcup(\mathbb{R}\times\mathbf{1})) = \mathcal{D}[\mathrm{Array}\,(\mathbb{R}\times\mathbb{Z})]_2$. Assuming the length of the array in x is n, this program contributes a cotangent of d to the $\mathbb{R}$ inside the array element at index 4 in x. Using **accum**, this can be written in any of the following three ways:

$$\mathbf{accum}_{(x:\mathbf{1}\sqcup\mathrm{Array}\,(\mathbf{1}\sqcup(\mathbb{R}\times\mathbf{1}))\in\Gamma),*}\ \langle\rangle\ (\mathrm{inr}\ (\mathrm{build}\ n\ (i.\ \mathbf{if}\ i = 4\ \mathbf{then}\ \mathrm{inr}\ \langle d, 0\rangle\ \mathbf{else}\ \mathrm{inl}\ \langle\rangle)))$$
$$\mathbf{accum}_{(x:\mathbf{1}\sqcup\mathrm{Array}\,(\mathbf{1}\sqcup(\mathbb{R}\times\mathbf{1}))\in\Gamma),!*}\ \langle 4, n, \langle\rangle\rangle\ (\mathrm{inr}\ \langle d, 0\rangle)$$
$$\mathbf{accum}_{(x:\mathbf{1}\sqcup\mathrm{Array}\,(\mathbf{1}\sqcup(\mathbb{R}\times\mathbf{1}))\in\Gamma),!\overleftarrow{\times}*}\ \langle 4, n, \langle\rangle\rangle\ d \qquad ③$$

As an example, the type of the first argument (namely $\langle 4, n, \langle\rangle\rangle$) to the invocation on line ③ is computed as follows:

$$\begin{aligned}
&\mathrm{APix}\ (!\overleftarrow{\times}*)\ (\mathbf{1}\sqcup\mathrm{Array}\,(\mathbf{1}\sqcup(\mathbb{R}\times\mathbf{1})))\\
&\quad= \mathbb{Z}\times\mathbb{Z}\times\mathrm{APix}\ (\overleftarrow{\times}*)\ (\mathbf{1}\sqcup(\mathbb{R}\times\mathbf{1}))\\
&\quad= \mathbb{Z}\times\mathbb{Z}\times\mathrm{APix}\ *\ \mathbb{R}\\
&\quad= \mathbb{Z}\times\mathbb{Z}\times\mathbf{1}
\end{aligned}$$

All three versions are semantically equivalent, since inl $\langle\rangle$ functions as the zero for the monoid $\mathcal{D}[\mathbb{R}\times\mathbb{Z}]_2 = \mathbf{1}\sqcup(\mathbb{R}\times\mathbf{1})$. However, the first version requires time $O(n)$ (and is operationally equivalent to the use of **one** above), whereas versions two and three run in $O(1)$ time if the cotangent array in x has already been initialised. The third version is the fastest.

Code transformation on IA. Firstly, with **one** replaced by **accum**, we have to rewrite the rule for variable references:

$$\mathcal{D}_\Gamma[x:\tau] = \langle x : \mathcal{D}[\tau]_1, \lambda d.\ \mathbf{accum}_{(x:\mathcal{D}[\tau]_2\in\mathcal{D}[\Gamma]_2),*}\ \langle\rangle\ d\rangle$$
$$\mathcal{D}_\Gamma[x:\tau] = \langle x : \mathcal{D}[\tau]_1, \lambda d.\quad \mathbf{one}_{x:\mathcal{D}[\tau]_2\in\mathcal{D}[\Gamma]_2}\quad d\rangle \qquad \text{(from Fig. 6.6)}$$

The version with **one** from Fig. 6.6 (page 246) is written in grey for comparison.

For the variable-only indexing construct in Idealised Accelerate, the following rule suffices:

$$\begin{aligned}&\mathcal{D}_\Gamma[(a : \text{Array } \tau)\ !\ t] = && (\text{`}a\text{' a variable reference})\\ &\quad \mathbf{let}\ n = \text{length}\ a; \langle i, \_\rangle = \mathcal{D}_\Gamma[t]\\ &\quad \mathbf{in}\ \langle a\ !\ i, \lambda d.\ \mathbf{accum}_{(a:\mathbf{1}\sqcup\text{Array}\,\mathcal{D}[\tau]_2\in\mathcal{D}[\Gamma]_2),!\,*}\ \langle i, n, \langle\rangle\rangle\ d\rangle\end{aligned}$$

This rule ensures that for terms that index variable references only, the derivative is always performed with a subobject accumulation, resulting in an $O(1)$ derivative of array indexing when the cotangent array is already allocated (see Section 7.2.2). For a perspective on the full source language, see below under 'Generalisation'.

The updated derivatives for 'build' and 'fold' are shown in Fig. 7.3. The major difference with Fig. 6.6 is that Bag $(\mathbb{Z}\times\mathcal{D}[\tau]_2)$ is replaced with $\mathbf{1}\sqcup\text{Array}\,\mathcal{D}[\tau]_2$; this simplifies $\mathcal{D}_\Gamma[\text{build}]$. For 'fold' we have done away with the difference list and reused our accumulator machinery to collect the cotangent for t in the reverse traversal of the Tree. In an implementation, one should use custom implementations of the primal and the dual of 'fold' that are specialised to the particular tree reduction structure that it actually performs, as already discussed in Section 6.7.1.

Generalisation. When expanding to the full source language, the unimaginative option is to give a fallback rule used whenever the first argument of (!) is not a variable reference:

$$\begin{aligned}&\mathcal{D}_\Gamma[s\ !\ t] =\\ &\quad \mathbf{let}\ \langle x, x'\rangle = \mathcal{D}_\Gamma[s]; \langle i, \_\rangle = \mathcal{D}_\Gamma[t]\\ &\quad \mathbf{in}\ \langle x\ !\ i, \lambda d.\ x'\ (\text{inr}\ (\text{build}\ (\text{length}\ x)\ (j.\ \mathbf{if}\ j = i\ \mathbf{then}\ d\ \mathbf{else}\ \underline{0})))\rangle\end{aligned}$$

However, while efficiently handling the full source language is future work (see Section 7.6), our more general definition of **accum** does allow us to do a little

$$
\begin{array}{l}
\mathcal{D}_\Gamma[\text{build } s\ (i.\ t : \tau)] = \\
\quad \textbf{let } \langle n, \_ \rangle = \mathcal{D}_\Gamma[s] \\
\quad\quad a = \text{build } n\ (i.\ \mathcal{D}_{\Gamma, i:\mathbb{Z}}[t]) \\
\quad\quad \langle a_1, a_2 \rangle = \text{unzip } a \\
\quad \textbf{in } \langle a_1, \lambda d.\ \textbf{case } d \textbf{ of} \\
\quad\quad\quad \text{inl } \langle\rangle \to \textbf{return } \langle\rangle \\
\quad\quad\quad \text{inr } da \to \textbf{do} \\
\quad\quad\quad\quad \text{sequence } (\text{zipWith } (f\ d'.\ \textbf{scope}_{\mathcal{D}[\Gamma]_2, \underline{\mathbf{1}}}\ (f\ d') \gg \textbf{return } \langle\rangle) \\
\quad\quad\quad\quad\quad\quad\quad\quad a_2\ da) \\
\quad\quad\quad\quad \textbf{return } \langle\rangle \rangle \\
\mathcal{D}_\Gamma[\text{fold } (p.\ s)\ (t : \text{Array } \tau)] = \\
\quad \textbf{let } \langle t_1, t_2 \rangle = \mathcal{D}_\Gamma[t] \\
\quad\quad tree = \text{fold } (p'.\ \textbf{let } p = \langle \text{getA } (\text{fst } p'), \text{getA } (\text{snd } p') \rangle \\
\quad\quad\quad\quad\quad\quad \langle y, f \rangle = \mathcal{D}_{\Gamma, p:\tau\times\tau}[s] \\
\quad\quad\quad\quad\quad\quad \textbf{in } \text{Node } (\text{fst } p')\ y\ f\ (\text{snd } p')) \\
\quad\quad\quad\quad (\text{build } (\text{length } t_1)\ (i.\ \text{Leaf } i\ (t_1\ !\ i))) \\
\quad \textbf{in } \langle \text{getA } tree \\
\quad\quad , \lambda d.\ \textbf{do } \langle \langle\rangle, da \rangle \leftarrow \textbf{scope}_{\mathcal{D}[\Gamma]_2, (da : \text{Array } \mathcal{D}[\tau]_2)} \\
\quad\quad\quad\quad\quad (\text{unTree } (\lambda d'\ f.\ \textbf{do} \\
\quad\quad\quad\quad\quad\quad\quad\quad \langle \langle\rangle, \langle d_1, d_2 \rangle \rangle \leftarrow \textbf{scope}_{(\mathcal{D}[\Gamma]_2, da), \mathcal{D}[\tau\times\tau]_2}\ (f\ d') \\
\quad\quad\quad\quad\quad\quad\quad\quad \textbf{return } \langle d_1, d_2 \rangle) \\
\quad\quad\quad\quad\quad\quad\quad\quad d\ tree\ (\lambda i\ d'.\ \textbf{accum}_{da, !\,*}\ \langle i, \text{length } t_1, \langle\rangle \rangle\ d')) \\
\quad\quad\quad\quad t_2\ (\text{inr } da) \rangle
\end{array}
$$

$$
\begin{array}{l}
\textbf{data } \text{Tree } a\ f = \text{Node } (\text{Tree } a\ f)\ a\ f\ (\text{Tree } a\ f) \mid \text{Leaf } \mathbb{Z}\ a \\
\text{getA} : \text{Tree } a\ f \to a \\
\text{getA } (\text{Node } \_\ x\ \_\ \_) = x \\
\text{getA } (\text{Leaf } \_\ x) = x \\
\text{unTree} : \text{Monad } m \Rightarrow (d \to f \to m\ (d \times d)) \to d \to \text{Tree } a\ f \\
\qquad\qquad \to (\mathbb{Z} \to d \to m\ \mathbf{1}) \to m\ \mathbf{1} \\
\text{unTree } g\ d\ (\text{Node } t_1\ \_\ f\ t_2)\ ac = \textbf{do } \langle d_1, d_2 \rangle \leftarrow g\ d\ f \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{unTree } g\ d_1\ t_1\ ac \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{unTree } g\ d_2\ t_2\ ac \\
\text{unTree } g\ d\ (\text{Leaf } i\ \_)\ ac = ac\ i\ d
\end{array}
$$

Figure 7.3: The 'build' and 'fold' operations with $\mathcal{D}[\text{Array } \tau]_2 = \mathbf{1} \sqcup \text{Array } \mathcal{D}[\tau]_2$ and **accum**. Differences with Fig. 6.6 (page 246) highlighted.

better by also considering a chain of product projection terms to be a suitable first argument to (!). We can achieve this compositionally by introducing a new target language term:

$$\mathbf{onehot}_{\tau,p} : \mathrm{APix}\ p\ \tau \to \mathrm{APty}\ p\ \tau \to \tau$$

such that, for example, $\mathbf{onehot}_{((\tau_1\times\tau_2)\times\tau_3),\overleftarrow{\times}\,\vec{\times}\,*}\ \langle\rangle\ x = \mathrm{inr}\ \langle \mathrm{inr}\ \langle \underline{0}_{\tau_1}, x\rangle, \underline{0}_{\tau_3}\rangle$ if x has type τ_2. The idea is that we will introduce post-AD rewrite rules that fuse **accum** and **onehot** to produce efficient accumulations whenever possible.

To make this work, we must update the rules for projection terms to produce **onehot** terms instead of directly constructing one-hot cotangents, for example:

$$\begin{aligned}
&\mathcal{D}_\Gamma[\mathrm{fst}\ (t : \sigma\times\tau)] = \\
&\quad \mathbf{let}\ \langle x, x'\rangle = \mathcal{D}_\Gamma[t] \\
&\quad \mathbf{in}\ \langle \mathrm{fst}\ x, \lambda d.\ x'\ (\mathbf{onehot}_{(\mathbf{1}\sqcup(\mathcal{D}[\sigma]_2\times\mathcal{D}[\tau]_2)),\overleftarrow{\times}\,*}\ \langle\rangle\ d)\rangle \\
&\mathcal{D}_\Gamma[(s : \mathrm{Array}\ \tau)\ !\ t] = \\
&\quad \mathbf{let}\ \langle x, x'\rangle = \mathcal{D}_\Gamma[s]; \langle i, _\rangle = \mathcal{D}_\Gamma[t] \\
&\quad \mathbf{in}\ \langle x\ !\ i, \lambda d.\ x'\ (\mathbf{onehot}_{(\mathbf{1}\sqcup\mathrm{Array}\ \mathcal{D}[\tau]_2),!\,*}\ \langle i, \mathrm{length}\ x, \langle\rangle\rangle\ d)\rangle
\end{aligned}$$

This is sufficient to ensure that a chain of product projections into a variable reference differentiates to a number of nested **onehot** calls passed to an **accum**. For example:

$$\begin{aligned}
&\mathcal{D}_\Gamma[\mathrm{fst}\ (a : \mathrm{Array}\ \tau\times\sigma)\ !\ t] \\
&= \mathbf{let}\ \langle x, x'\rangle = \mathbf{let}\ \langle y, y'\rangle = \langle a, \lambda d.\ \mathbf{accum}_{(a:\mathbf{1}\sqcup((\mathbf{1}\sqcup\mathrm{Array}\ \mathcal{D}[\tau]_2)\times\mathcal{D}[\sigma]_2)\in\mathcal{D}[\Gamma]_2),*}\ \langle\rangle\ d\rangle \\
&\qquad\qquad \mathbf{in}\ \langle \mathrm{fst}\ y, \lambda d.\ y'\ (\mathbf{onehot}_{(\mathbf{1}\sqcup((\mathbf{1}\sqcup\mathrm{Array}\ \mathcal{D}[\tau]_2)\times\mathcal{D}[\sigma]_2)),\overleftarrow{\times}\,*}\ \langle\rangle\ d)\rangle \\
&\qquad \langle i, _\rangle = \mathcal{D}_\Gamma[t] \\
&\quad \mathbf{in}\ \langle x\ !\ i, \lambda d.\ x'\ (\mathbf{onehot}_{(\mathbf{1}\sqcup\mathrm{Array}\ \mathcal{D}[\tau]_2),!\,*}\ \langle i, \mathrm{length}\ x, \langle\rangle\rangle\ d)\rangle \\
&= \langle \mathrm{fst}\ a\ !\ \mathrm{fst}\ \mathcal{D}_\Gamma[t] \\
&\quad , \lambda d.\ \mathbf{accum}_{(a:\mathbf{1}\sqcup((\mathbf{1}\sqcup\mathrm{Array}\ \mathcal{D}[\tau]_2)\times\mathcal{D}[\sigma]_2)\in\mathcal{D}[\Gamma]_2),*}\ \langle\rangle \\
&\qquad\qquad (\mathbf{onehot}_{(\mathbf{1}\sqcup((\mathbf{1}\sqcup\mathrm{Array}\ \mathcal{D}[\tau]_2)\times\mathcal{D}[\sigma]_2)),\overleftarrow{\times}\,*}\ \langle\rangle \\
&\qquad\qquad\quad (\mathbf{onehot}_{(\mathbf{1}\sqcup\mathrm{Array}\ \mathcal{D}[\tau]_2),!\,*}\ \langle i, \mathrm{length}\ x, \langle\rangle\rangle\ d)))\rangle
\end{aligned}$$

Combinations of **accum** and **onehot**, and nested occurrences of **onehot**, can be fused into a single operation by composing the projections. This fusion can be accomplished in the context of a standard simplifier by local rewriting. In this case, the result is the following term with a single accumulation and no remaining one-hots:

$$\begin{aligned}
&\langle \mathrm{fst}\ a\ !\ \mathrm{fst}\ \mathcal{D}_\Gamma[t] \\
&, \lambda d.\ \mathbf{accum}_{(a:\mathbf{1}\sqcup((\mathbf{1}\sqcup\mathrm{Array}\ \mathcal{D}[\tau]_2)\times\mathcal{D}[\sigma]_2)\in\mathcal{D}[\Gamma]_2),\overleftarrow{\times}\,!\,*}\ \langle i, \mathrm{length}\ x, \langle\rangle\rangle\ d\rangle
\end{aligned}$$

where the projection '$\overleftarrow{\times}\,!\,*$' results from composing '$*$', '$\overleftarrow{\times}\,*$' and '$!\,*$'.

This fusion system may be extended by intelligently merging **onehot** with array operations; for example, indexing into a **onehot** value can be converted

to a conditional, and summing the values in a **onehot** array can return just the non-zero element.

If uses of **onehot** remain after this fusion process, the program performed indexing into a term that is not a projection of a variable reference; in this case, we currently propose no better solution than simply constructing the full one-hot.[10]

7.2.2 Non-constant-time zeros

While the inclusion of sparsity in $\mathcal{D}[\text{Array } \tau]_2 = \mathbf{1} \sqcup \text{Array } \mathcal{D}[\tau]_2$ means that array cotangents technically have an $O(1)$ zero, this is insufficient when we need to accumulate individual element cotangents to an array cotangent. Strictly interpreting the definitions in this section so far, **accum** will need to expand the inl $\langle\rangle$ zero value of a $\mathcal{D}[\text{Array } \tau]_2$ into a fully-allocated array of $\underline{0}_{\mathcal{D}[\tau]_2}$ upon the first (non-zero) accumulation. This behaviour is quite awkward, especially on large vector machines: it entails an additional conditional in the dual of every array indexing operation, as well as a lock on the array cotangent to let other (parallel) accumulations wait until the array has been allocated. Thus, for array cotangent zeros in sites that will be subobject-accumulated into (i.e. the initialiser of **scope**), it makes more sense to allocate a dense array of (possibly sparse) zeros a priori, so that any new allocations or locks are at least thread-local afterwards.[11]

Regardless of whether we allocate the array zero in **scope** or later in **accum**, however, most uses of **scope** on a type that includes arrays will now lead to an array zero allocation (directly or indirectly). As **scope** represents the introduction of a variable in a binding construct in the source language, and such introductions represent only $O(1)$ work, allocating a dense zero array breaks our complexity guarantees. Indeed, amortisation cannot save us here: the derivative of this term:

$$\textbf{let } a_1 = \text{build } \ldots \textbf{ in let } a_2 = a_1 \textbf{ in } \ldots \textbf{ in let } a_n = a_{n-1} \textbf{ in } \ldots$$

binds the same array n times, resulting in n zero allocations.[12]

For Futhark, Schenck et al. [2022] ignore zero management and costs in writing, as they do not try to achieve optimal complexity. In their implementation, however, they do use a clever trick to reduce the number of array zeros for accumulation: whenever a cotangent accumulator is to be created for a certain source array, and there is already a cotangent contribution for that source array

[10] However, we conjecture that this can be significantly improved; see Section 7.6.

[11] Note that we cannot drop the sparsity in $\mathcal{D}[\text{Array } \tau]_2$ altogether lest $\underline{0}_{\mathcal{D}[\text{Array } \tau]_2}$ become expensive; doing so would impact all other uses of $\underline{0}$ in the algorithm, such as in $\mathcal{D}_\Gamma[\text{fst } t]$.

[12] If the zeros are allocated by **accum**, add one use of indexing to each let-binding. If one objects that a compiler will simplify such a term: make the right-hand sides non-trivial, use a_i multiple times, and add generous amounts of dynamic control flow.

known in the algorithm, that existing contribution is used as the initialiser of the accumulator in **scope** instead of a zero. That is to say, instead of:

$$\begin{aligned}&\textbf{do } \langle \langle\rangle, da_2\rangle \leftarrow \textbf{scope}'_{\mathcal{D}[\Gamma]_2, \mathcal{D}[\text{Array } \tau]_2}\ \underline{0}_{\mathcal{D}[\text{Array } \tau]_2}\ (\mathcal{D}_{\Gamma, a:\text{Array } \tau}[\ldots])\\ &\quad \textbf{return } (\text{zipWith } (+)\ da_1\ da_2)\end{aligned}$$

do the following:

$$\begin{aligned}&\textbf{do } \langle \langle\rangle, da\rangle \leftarrow \textbf{scope}'_{\mathcal{D}[\Gamma]_2, \mathcal{D}[\text{Array } \tau]_2}\ da_1\ (\mathcal{D}_{\Gamma, a:\text{Array } \tau}[\ldots])\\ &\quad \textbf{return } da\end{aligned}$$

where **scope′** is a variant of **scope** that allows an explicit initialiser (instead of zero). If da_1 is dense and known to be used only here, we can mutate it directly without copying and thus save a zero array and an elementwise addition. The restriction that da_1 is not used elsewhere is enforced in Futhark using its uniqueness type system.

This scheme is not complete, however; with application of a sufficient amount of dynamic control flow, the compiler can be made to miss the fact that two accumulator contributions are actually to the same array; in this case, multiple zeros are still generated.

Our prototype implementation has a simpler (but even less powerful) scheme for avoiding redundant zeros: when opening a **scope** for an array that is provably the same as a **scope** that is already open (as determined using a data-flow analysis), the outer **scope** is reused.[13] This seems to be sufficient to avoid most redundant zeros with non-pathological input program structures.

7.3 Inlining backpropagator lambdas

After addressing one of the sparsity-related performance issues in the algorithm (see Sections 7.5.1.2 and 7.6 for further progress and perspectives on this topic), we continue with an optimisation of a different nature: eliminating unnecessary higher-order code from the derivative program. Indeed, the output of CHAD is replete with lambda abstractions: every term former in the source program gets its own backpropagator. Lambda abstractions are normally bad news for performance: when not simplified away in optimisation, they inhibit compiler optimisations by disrupting program flow locality, they may result in heap allocations at runtime for their closure, and they require an indirect jump instruction to call. It would be quite unfortunate if even some of these backpropagators remain after simplification in the program to be compiled. And in fact, high-performance

[13] By splitting let-bindings of products into individual let-bindings, this can work even if the array in question is bound as part of a tuple by the programmer. See also Section 7.5.1.4.

array language compilers in practice simply do not support higher-order functions, for these and other reasons.

Fortunately, such expressivity of the target language is unnecessary, as we can do entirely without explicit lambda abstractions for backpropagators.[14] To see this, we start by observing that in almost all rules of the transformation, backpropagator functions are applied in *exactly one place* in the derivative program. Note that this statement is unrelated to any claim about how many times they are applied at runtime; indeed, the complexity proof of Chapter 6 already shows that every backpropagator is applied at most once at runtime, but this by itself does not put any bound on the number of function-application *terms* in the derivative program that apply a particular backpropagator.

As an example, consider the rule for pairs in Fig. 6.6 (page 246):

$$\begin{aligned}\mathcal{D}_\Gamma[\langle s, t\rangle] = \mathbf{let}\ \langle x, x'\rangle = \mathcal{D}_\Gamma[s]; \langle y, y'\rangle = \mathcal{D}_\Gamma[t] \\ \mathbf{in}\ \langle\langle x, y\rangle, \lambda d.\ \mathbf{do}\ x'\ (\mathbf{lfst}\ d); y'\ (\mathbf{lsnd}\ d)\rangle\end{aligned} \tag{7.2}$$

Two backpropagators are created here (x' and y'), and both are applied in exactly one textual location in the derivative program. Furthermore, we know statically which location that is. As a result, under the assumption that the bodies of x' and y' can be cleanly extracted from $\mathcal{D}_\Gamma[s]$ and $\mathcal{D}_\Gamma[t]$, inlining them at their application sites on the second line does not increase code size and would accomplish our goal of removing their lambda abstractions.

It turns out that this "clean extraction" is always possible for the CHAD rules presented in Chapter 6. Specifically, each rule is either already in the following form or can be converted to it:

$$\mathcal{D}_\Gamma[t] = \mathbf{let}\ \mathit{bindings...}\ \mathbf{in}\ \langle \mathit{primal}, \lambda d.\ \mathit{dual}\rangle \tag{7.3}$$

At a high level, this means that CHAD neatly falls apart into a "forward pass" ('*bindings...*' plus '*primal*') and a "reverse pass" ('*dual*'), as desired and expected for a performant reverse AD algorithm. For terms that do not involve any kind of control flow (or iteration), the rules are already in the indicated form and no work is necessary to split them into the indicated three parts and thus extricate the body of the backpropagator, i.e. the reverse pass. For more involved derivatives, like those of **case** and 'build', some amount of manipulation is required to achieve the splitting without introducing additional closures; the necessary changes here can be interpreted as an instance of closure conversion (or defunctionalisation), as we need to split some closures into a closed function and an explicit list of its free variables.

[14] This was already foreshadowed on page 222. Naturally, if the source program already contained closures, those will still be present in the differentiated program.

In contrast to Section 4.6, where we were satisfied with an external proof that Delta term extraction always works so that we could express the pass as term interpretation again, here we split up the code transformation into three parts (bindings, primal and dual) to ensure extractibility of the reverse pass by construction. Aside from eliminating lambda abstractions, this reformulation also makes some data flow in the derivative program more explicit; this surfaces additional improvements to be made, one described explicitly (Section 7.4), others implemented but only sketched in this thesis (Section 7.5.1). We start with definitions and simple cases in Section 7.3.1, after which we discuss and adapt terms with control flow separately in Section 7.3.2.

The sufficiently smart compiler. In Chapter 5 when discussing the example on page 222, we noted that simple compiler optimisations could already inline all backpropagators in the derivative program, leaving a completely first-order result. Thus, the reader may wonder whether this is always possible, and thus whether the explicit removal of backpropagator lambdas that we accomplish in this section is in fact redundant: assuming a sufficiently smart compiler, the "external proof that it always works" strategy may be able to save us some work.

If the input program consists only of straight-line, scalar-only code (as the example in Chapter 5 does), then a standard, uncomplicated inlining optimisation achieves the same performance as the split transformation in this section. (Indeed, the rules for such term formers are trivially split in Section 7.3.1.) However, in the presence of dynamic control flow (exemplified by **case** in Fig. 6.6 on page 246) or arrays ('build' in Fig. 6.8 on page 262), the required compiler smartness level increases significantly, including needing to track closures through values in an array whose size is known only at runtime. Thus, in addition to the benefits of enabling other optimisations (such as Section 7.4) and being applicable to languages whose compilers do not support lambda abstraction at all, we also expect observable performance benefits in practice.

7.3.1 Split code transformation

To formalise the splitting sketched by Eq. (7.3) on page 317, we define three mutually inductive code transformations $\mathcal{D}^0_\Gamma[t], \mathcal{D}^1_\Gamma[t], \mathcal{D}^2_{\Gamma,d}[t]$. The guiding design principle is the following result:

Theorem 3. *We have the following semantic equality:*

$$[\![\mathcal{D}_\Gamma[t]]\!] = [\![\textbf{let } \mathcal{D}^0_\Gamma[t] \textbf{ in } \langle \mathcal{D}^1_\Gamma[t], \lambda d.\ \mathcal{D}^2_{\Gamma,d}[t] \rangle]\!]$$

The proof is by induction on t after we have defined the rules for the new code transformations, and can be assembled from individual justifications throughout this section.

Notation:

$$\Gamma, |\{x_1 : \tau_1 = t_1, \ldots, x_n : \tau_n = t_n\}| = \Gamma, x_1 : \tau_1, \ldots, x_n : \tau_n$$

$$\{x_1 : \sigma_1 = s_1, \ldots, x_n : \sigma_n = s_n\} +\!\!+ \{y_1 : \tau_1 = t_1, \ldots, y_n : \tau_n = t_n\} = \{x_1 : \sigma_1 = s_1, \ldots, x_n : \sigma_n = s_n, y_1 : \tau_1 = t_1, \ldots, y_n : \tau_n = t_n\}$$

Typing rules:

$$\frac{}{\Gamma \vdash_{\text{binds}} \{\}} \qquad \frac{\Gamma \vdash_{\text{binds}} \{bs\} \quad \Gamma, |\{bs\}| \vdash t : \tau}{\Gamma \vdash_{\text{binds}} \{bs, x : \tau = t\}} \qquad \frac{\Gamma \vdash_{\text{binds}} \{bs\} \quad \Gamma, |\{bs\}| \vdash t : \tau}{\Gamma \vdash \textbf{let}\ \{bs\}\ \textbf{in}\ t : \tau}$$

Figure 7.4: Ordered binding lists. The type "$: \tau$" in a binding can be elided when clear from context. The idea is that **let** $\{x = s, y = t\}$ **in** u means **let** $x = s$ **in let** $y = t$ **in** u. 'bs' above is short for any finite number of bindings of the form $x : \tau = t$.

Firstly, for the notation in the theorem to make sense, we must have a concept of a "list of let-bindings" that can be produced by a syntactic transformation. To that end, we add *binding lists* to the target language, with the typing shown in Fig. 7.4.[15] Note that this addition does not increase the expressivity of the target language in any way; it is just a syntactic device that allows us to conveniently specify $\mathcal{D}^0_\Gamma[t]$.

With this new notation, we can specify the meta-types of the three split-up parts of $\mathcal{D}_\Gamma[t]$:

$$\begin{aligned} &\mathcal{D}[\Gamma]_1 \vdash_{\text{binds}} \mathcal{D}^0_\Gamma[t] \\ &\mathcal{D}[\Gamma]_1, |\mathcal{D}^0_\Gamma[t]| \vdash \mathcal{D}^1_\Gamma[t] : \mathcal{D}[\tau]_1 \\ &\mathcal{D}[\Gamma]_1, |\mathcal{D}^0_\Gamma[t]|, d : \mathcal{D}[\tau]_2 \vdash \mathcal{D}^2_{\Gamma,d}[t] : \textbf{EVM}\ \mathcal{D}[\Gamma]_2\ \mathbf{1} \end{aligned} \tag{7.4}$$

These are, respectively, the shared bindings, the primal and the backpropagator (dual / cotangent computation) of the derivative of t. The additional index d of $\mathcal{D}^2[t]$ is the name of a variable that contains the incoming cotangent for t. This variable name takes the role of the function argument to the backpropagator in the CHAD transformations of Chapters 5 and 6.

For example, the definitions for a pair term (compare Eq. (7.2) on page 317)

[15] Note that binding lists are ordered despite the {} notation, which we choose to be visually distinctive from [] in the figures and equations in this chapter.

are as follows:[16]

$$\begin{aligned}
\mathcal{D}^0_\Gamma[\langle s,t\rangle] &= \mathcal{D}^0_\Gamma[s] +\!\!+ \mathcal{D}^0_\Gamma[t] +\!\!+ \{x = \mathcal{D}^1_\Gamma[s], y = \mathcal{D}^1_\Gamma[t]\}\\
\mathcal{D}^1_\Gamma[\langle s,t\rangle] &= \langle x, y\rangle\\
\mathcal{D}^2_{\Gamma,d}[\langle s,t\rangle] &= \mathbf{do}\ (\mathbf{let}\ d' = \mathbf{lfst}\ d\ \mathbf{in}\ \mathcal{D}^2_{\Gamma,d'}[s])\\
&\qquad\qquad (\mathbf{let}\ d' = \mathbf{lsnd}\ d\ \mathbf{in}\ \mathcal{D}^2_{\Gamma,d'}[t])
\end{aligned}$$

Equivalently, we could reorder the bindings in $\mathcal{D}^0_\Gamma[\langle s,t\rangle]$:

$$\mathcal{D}^0_\Gamma[\langle s,t\rangle] = \mathcal{D}^0_\Gamma[s] +\!\!+ \{x = \mathcal{D}^1_\Gamma[s]\} +\!\!+ \mathcal{D}^0_\Gamma[t] +\!\!+ \{y = \mathcal{D}^1_\Gamma[t]\}$$

Notable is that regardless of which precise order the bindings have, the three-part form of this new transformation forces us to flatten out the lists of bindings of the two subterms into one long list. In the original $\mathcal{D}_\Gamma[\langle s,t\rangle]$ (Eq. (7.2) or Fig. 6.6), the bindings for s and t that we inlined here were nested on the right-hand sides of the let-bindings for $\langle x, x'\rangle$ and $\langle y, y'\rangle$.

Furthermore, note that we apply some significant implicit weakenings in $\mathcal{D}^0_\Gamma[\langle s,t\rangle]$ and $\mathcal{D}^2_{\Gamma,d}[\langle s,t\rangle]$. There are already implicit weakenings in Eq. (7.2): strictly speaking, $\mathcal{D}_\Gamma[t]$ is a term in environment $\mathcal{D}[\Gamma]_1$ whereas we use it in the extended environment $\mathcal{D}[\Gamma]_1, x : \mathcal{D}[\sigma]_1, x' : \mathcal{D}[\sigma]_2 \to \mathbf{EVM}\ \mathcal{D}[\Gamma]_2\ \mathbf{1}$.[17] Similarly, in $\mathcal{D}^0_\Gamma[\langle s,t\rangle]$, all bindings in $\mathcal{D}^0_\Gamma[t]$ must be weakened under all bindings of $\mathcal{D}^0_\Gamma[s]$, with analogous weakenings for $\mathcal{D}^1_\Gamma[s]$ and $\mathcal{D}^1_\Gamma[t]$; in $\mathcal{D}^2_{\Gamma,d}[\langle s,t\rangle]$, $\mathcal{D}^2_{\Gamma,d'}[s]$ has environment $\mathcal{D}[\Gamma]_1, |\mathcal{D}^0_\Gamma[s]|, d' : \mathcal{D}[\sigma]_2$, but we use it in an environment with an additional, ignored $d : \mathcal{D}[\sigma \times \tau]_2$.

In an implementation, especially one based on De Bruijn indices, these weakenings must be made explicitly, but because the weakenings to apply should always be clear from context, we elide them on paper.

Other straight-line terms. The other term formers without control flow work completely analogously to the pair term considered so far. For example, for let-binding we have: (compare Fig. 6.6 on page 246 again)

$$\begin{aligned}
\mathcal{D}^0_\Gamma[\mathbf{let}\ x : \tau = s\ \mathbf{in}\ t] &= \mathcal{D}^0_\Gamma[s] +\!\!+ \{x = \mathcal{D}^1_\Gamma[s]\} +\!\!+ \mathcal{D}^0_{\Gamma,x:\tau}[t]\\
\mathcal{D}^1_\Gamma[\mathbf{let}\ x : \tau = s\ \mathbf{in}\ t] &= \mathcal{D}^1_{\Gamma,x:\tau}[t]\\
\mathcal{D}^2_{\Gamma,d}[\mathbf{let}\ x : \tau = s\ \mathbf{in}\ t] &= \mathbf{do}\ \langle\langle\rangle, dx\rangle \leftarrow \mathbf{scope}_{\mathcal{D}[\Gamma]_2, \mathcal{D}[\tau]_2}\ \mathcal{D}^2_{(\Gamma,x:\tau),d}[t]\\
&\qquad\qquad \mathcal{D}^2_{\Gamma,dx}[s]
\end{aligned}$$

Note that in contrast to $\mathcal{D}^0_\Gamma[\langle s,t\rangle]$, the ordering of the bindings in $\mathcal{D}^0_\Gamma[\mathbf{let}]$ *is* important, as we must put $\mathcal{D}^0_{\Gamma,x:\tau}[t]$ after the binding $\{x = \mathcal{D}^1_\Gamma[s]\}$ that it refers to.

[16] We maintain the convention from Chapter 5 (on page 209) that variables appearing on the right-hand side and not on the left-hand side, like x and y here, are fresh.

[17] We drop the distinction between $\multimap$ and $\to$ in this chapter as there is no operational difference.

7.3.2 Control flow

Something more interesting must be done for source terms that perform some kind of control flow. The quintessential example is **case**, and looking at Fig. 6.6, we see that it violates the principle that the backpropagator of a subterm is invoked in exactly one location in the derivative term: z' is applied in two locations. Simply inlining the backpropagator for s would thus result in a exponential code size blowup (if **case** is nested), which is undesirable. Even worse, because the derivative is not of the usual form (**let** *bindings...* **in** $\langle primal, \lambda d.\ dual \rangle$), we cannot systematically extract suitable definitions for $\mathcal{D}^1_\Gamma$ and $\mathcal{D}^2_{\Gamma,d}$ at all.

Fortunately, fixing the latter problem also fixes the former problem. The trick is to rewrite $\mathcal{D}_\Gamma[\mathbf{case}]$ to "export" from the derivative **case** all values necessary to construct the primal and the backpropagator for the entire source **case**, and then have a single pair term at the end where we perform said construction. With the plain $\mathcal{D}_\Gamma$ transformation of Chapter 6, we could technically achieve the stated goals as follows:

$$
\begin{aligned}
&\mathcal{D}_\Gamma[\mathbf{case}\ s : \sigma \sqcup \tau\ \mathbf{of}\ \{\ \mathrm{inl}\ x \to t_1 \mid \mathrm{inr}\ y \to t_2\ \}] = \\
&\quad \mathbf{let}\ \langle z, z' \rangle = \mathcal{D}_\Gamma[s] \\
&\qquad \langle r, r' \rangle = \mathbf{case}\ z\ \mathbf{of} \\
&\qquad\qquad \mathrm{inl}\ x \to \mathbf{let}\ \langle x_1, x_2 \rangle = \mathcal{D}_{\Gamma, x:\sigma}[t_1] \\
&\qquad\qquad\qquad\qquad \mathbf{in}\ \langle x_1, \mathrm{inl}\ (x_2 : \mathcal{D}[\sigma]_2 \to \mathbf{EVM}\ \mathcal{D}[\Gamma, x : \sigma]_2\ \mathbf{1}) \rangle \\
&\qquad\qquad \mathrm{inr}\ y \to \mathbf{let}\ \langle y_1, y_2 \rangle = \mathcal{D}_{\Gamma, y:\tau}[t_2] \\
&\qquad\qquad\qquad\qquad \mathbf{in}\ \langle y_1, \mathrm{inr}\ (y_2 : \mathcal{D}[\tau]_2 \to \mathbf{EVM}\ \mathcal{D}[\Gamma, y : \tau]_2\ \mathbf{1}) \rangle \\
&\quad \mathbf{in}\ \langle r, \lambda d.\ \mathbf{do}\ dz' \leftarrow \mathbf{case}\ r'\ \mathbf{of} \\
&\qquad\qquad\qquad\qquad \mathrm{inl}\ x_2 \to \mathbf{do}\ \langle \langle \rangle, dz \rangle \leftarrow \mathbf{scope}_{\mathcal{D}[\Gamma]_2, \mathcal{D}[\sigma]_2}\ (x_2\ d) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{return}\ (\mathbf{linl}\ dz) \\
&\qquad\qquad\qquad\qquad \mathrm{inr}\ y_2 \to \mathbf{do}\ \langle \langle \rangle, dz \rangle \leftarrow \mathbf{scope}_{\mathcal{D}[\Gamma]_2, \mathcal{D}[\tau]_2}\ (y_2\ d) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{return}\ (\mathbf{linr}\ dz) \\
&\qquad\qquad\qquad z'\ dz' \rangle
\end{aligned}
$$

However, while this does put the tuple at the bottom of the derivative term and ensure that every backpropagator is called in exactly one location, it is not usefully convertible to the three-part form of the new transformation: we would have to pack $\mathcal{D}^2_{(\Gamma,x:\sigma),\_}[t_1]$ and $\mathcal{D}^2_{(\Gamma,y:\tau),\_}[t_2]$ in closures to be able to store them in r', and that is exactly what we are trying to avoid.

The solution is to store not the backpropagators for the reverse pass of the **case**, but instead the contents of their environments: this allows us to inline their code later after restoring that environment. In essence, we perform a specialised, local defunctionalisation of the backpropagators of the branches.

First, we introduce notation for a tuple of the variables bound in an environ-

ment:

$$\overline{x_1 : \tau_1, \ldots, x_n : \tau_n} \coloneqq \langle x_1, \ldots, x_n \rangle$$

The precise structure of the pairs that make up this tuple is unimportant. We also write simply $\overline{bs} \coloneqq |\overline{bs}|$ if bs is a binding list.

Now, we can write down the three-part derivative of **case**. For brevity, write CASE for the term "**case** $s : \sigma \sqcup \tau$ **of** $\{$ inl $x \to t_1 \mid$ inr $y \to t_2$ $\}$".[18]

$$\begin{aligned}
\mathcal{D}^0_\Gamma[\text{CASE}] = \mathcal{D}^0_\Gamma[s] \mathbin{+\!\!+}& \\
\{\langle r, \mathit{tape}\rangle = \textbf{case } \mathcal{D}^1_\Gamma[s] \textbf{ of}& \\
&\text{inl } x \to \textbf{let } \mathcal{D}^0_{\Gamma,x:\sigma}[t_1] \\
&\qquad \textbf{in } \langle \mathcal{D}^1_{\Gamma,x:\sigma}[t_1], \text{inl } \langle x, \overline{\mathcal{D}^0_{\Gamma,x:\sigma}[t_1]}\rangle\rangle \\
&\text{inr } y \to \textbf{let } \mathcal{D}^0_{\Gamma,y:\tau}[t_2] \\
&\qquad \textbf{in } \langle \mathcal{D}^1_{\Gamma,y:\tau}[t_2], \text{inr } \langle y, \overline{\mathcal{D}^0_{\Gamma,y:\tau}[t_2]}\rangle\rangle\} \\
\mathcal{D}^1_\Gamma[\text{CASE}] = r& \\
\mathcal{D}^2_{\Gamma,d}[\text{CASE}] = \textbf{do } d'' \leftarrow \textbf{case } \mathit{tape} \textbf{ of}& \\
&\text{inl } \langle x, \overline{\mathcal{D}^0_{\Gamma,x:\sigma}[t_1]}\rangle \to \\
&\quad \textbf{do } \langle\langle\rangle, d'\rangle \leftarrow \textbf{scope}_{\mathcal{D}[\Gamma]_2, \mathcal{D}[\sigma]_2}\ (\mathcal{D}^2_{(\Gamma,x:\sigma),d}[t_1]) \\
&\qquad \textbf{return } (\textbf{linl } d') \\
&\text{inr } \langle y, \overline{\mathcal{D}^0_{\Gamma,y:\tau}[t_2]}\rangle \to \\
&\quad \textbf{do } \langle\langle\rangle, d'\rangle \leftarrow \textbf{scope}_{\mathcal{D}[\Gamma]_2, \mathcal{D}[\tau]_2}\ (\mathcal{D}^2_{(\Gamma,y:\tau),d}[t_2]) \\
&\qquad \textbf{return } (\textbf{linr } d') \\
&\mathcal{D}^2_{\Gamma,d''}[s]
\end{aligned}$$

In the forward pass, we replace the backpropagator with not only the bindings of the branches but also the contents of the scrutinee (x or y as appropriate). In the reverse pass, we then perform a second conditional branch to simultaneously select which branch to backpropagate through (t_1 or t_2) and to restore the primal bindings necessary for that branch as well as the scrutinee. We store x and y here because the contexts of $\mathcal{D}^2_{(\Gamma,x:\sigma),d}[t_1]$ and $\mathcal{D}^2_{(\Gamma,y:\tau),d}[t_2]$ respectively include $\mathcal{D}[\Gamma, x : \sigma]_1 = \mathcal{D}[\Gamma]_1, x : \mathcal{D}[\sigma]_1$ and $\mathcal{D}[\Gamma, y : \tau]_1 = \mathcal{D}[\Gamma]_1, y : \mathcal{D}[\tau]_1$. The rest of the primal environment of the branches, i.e. $\mathcal{D}[\Gamma]_1$, is inherited from $\mathcal{D}^2_{\Gamma,d}[\text{CASE}]$.[19]

Complexity. The complexity argument in Chapter 6 worked because every transformation rule had proportional costs on the left and right sides except the

[18] Note that this *tape* is *not* linear as in taping reverse AD (Section 2.2.8): it has structure mirroring the lexical structure of the source program. It is a data structure that stores 1. all primal values, and 2. the control flow path taken; the latter dictates precisely what values are stored. Branching is reflected as a coproduct, as shown; a sequential loop would be reflected as an array or list.

[19] Preserving $\mathcal{D}[\Gamma]_1$ actually turns out to be unnecessary; we address this in Section 7.4.

rule for a variable reference, and the expensive addition operation performed there could be amortised over construction of cotangents elsewhere. While the concessions made in Section 7.2 mean that our complexity can not be called optimal any more, it seems that our new rule for **case** is also problematic: it performs $O(n)$ extra work in constructing and reading the *tape* tuple, where n is the number of bindings in $\mathcal{D}^0_{\Gamma,\_}[t_i]$. Fortunately, this cost can be amortised in a way not dissimilar to how we handled variable references:[20]

- Every binding made in $\mathcal{D}^0$ can be matched up with distinct $O(1)$ work done in the source program (clear by inspection of Figs. 7.5 to 7.7 below; note that the number of $\mathcal{D}^0$ bindings for any particular term former is bounded);
- Every $\mathcal{D}^0$ binding is stored at most *once* by a **case** derivative: when **case** is nested, bindings from within an inner **case** appear only in the single *tape* binding from that inner **case**;
- Thus the total work of storing all these *tape* tuples (in all **case** derivatives in the program) is at most proportional to the total number of bindings put in $\mathcal{D}^0$ in the program, which is at most proportional to the total amount of work in the source program.

Thus, switching to this **case** derivative raises the overhead over the source program by only a constant factor, which preserves asymptotic complexity.

Array combinators. The second-order array combinator derivatives in Fig. 7.3 (page 313) can be adapted to the three-part transformation in a fashion analogous to **case**; the results are shown in Section 7.3.3 (Figs. 7.6 and 7.7).

Specifically: for 'build', instead of generating backpropagator closures in the forward pass (in a_2 in Fig. 7.3), Fig. 7.6 stores tuples of primal bindings in *tapes*, restores them in the 'build' in the dual using indexing, and inlines $\mathcal{D}^2_{(\Gamma,i:\mathbb{Z}),d'}[t]$ where Fig. 7.3 (and Fig. 6.8) had a call to f. Note that the bindings in $\mathcal{D}^0_{\Gamma,i:\mathbb{Z}}[t]$ are known statically, and in particular their types do not depend on the value of i; this means that *tapes* is indeed a homogeneous array and thus typechecks.

For 'fold' (Fig. 7.7), the f field in the Tree, which stored backpropagators before, now stores primal binding tuples. These tuples are generated in the forward pass and restored in the function passed to 'unTree', with $\mathcal{D}^2_{(\Gamma,p:\tau\times\tau),d'}[s]$ inlined in **scope** where Fig. 7.3 (and Fig. 6.8) had a call to f. Note that to ensure that p is properly in scope again for $\mathcal{D}^2_{(\Gamma,p:\tau\times\tau),d'}[s]$, we need to augment unTree to also provide primals.

[20] One can formalise this argument by giving the source language (temporarily for the proof) a cost model that is k times the original cost model, for $k \approx 3$ (justified by scaling c in Section 6.5 (page 253) by k). These additional work tokens can then be spent on storing the *tape* tuples.

Finally, array indexing (Fig. 7.6) fits the basic pattern and is trivially converted.

For 'build' and 'fold', an argument analogous to that for **case** above shows that storing and retrieving the tuples in *tapes* and *tree* is admissible in our complexity.

7.3.3 Full transformation

The full split transformation is given in Figs. 7.5 to 7.7. The rules are based on Section 7.2, with (!), 'build', 'fold', 'fst' and 'snd' given there and other rules taken unchanged from Fig. 6.6 (page 246). To be noted is that all generated code (for our first-order input language[21]) is now first-order by construction, in contrast to all previous versions of the algorithm. Furthermore, we hope it is clear that the translation from the previous rules to Figs. 7.5 to 7.7 is largely mechanical except for the local defunctionalisation applied in $\overline{\mathcal{D}_\Gamma}[\mathbf{case}]$, $\overline{\mathcal{D}_\Gamma}[\text{build}]$ and $\overline{\mathcal{D}_\Gamma}[\text{fold}]$, which we discussed above.

Let us repeat the example $a : \mathbb{R} \vdash t_1 : \mathbb{R}$ from pages 218 to 222 in Section 5.1.5:

$$\begin{aligned} t_1 = \; & \mathbf{let}\ b = \langle a, 2 \cdot a \rangle \\ & \mathbf{in}\ \text{fst}\ b + \text{snd}\ b \end{aligned}$$

We get:

$$\begin{aligned}
&\mathbf{let}\ \mathcal{D}^0_{a:\mathbb{R}}[t_1]\ \mathbf{in}\ \langle \mathcal{D}^1_{a:\mathbb{R}}[t_1], \lambda d.\ \mathcal{D}^2_{(a:\mathbb{R}),d}[t_1] \rangle \\
&\quad = \mathbf{let}\ m_1 = 2 \\
&\qquad\quad m_2 = a \qquad\qquad (m_i\text{: from } \mathcal{D}^0_{a:\mathbb{R}}[2 \cdot a]\text{, renamed from } x_i \text{ in Fig. 7.6}) \\
&\qquad\quad b = \langle a, 2 \cdot a \rangle \qquad (b\text{: from } \mathcal{D}^0_{a:\mathbb{R}}[\mathbf{let}\ b = \ldots\ \mathbf{in}\ \ldots]) \\
&\qquad\quad x_1 = \text{fst}\ b \\
&\qquad\quad x_2 = \text{snd}\ b \qquad\qquad (x_i\text{: from } \mathcal{D}^0_{a:\mathbb{R},b:\mathbb{R}\times\mathbb{R}}[\text{fst}\ b + \text{snd}\ b]) \\
&\quad\ \mathbf{in}\ \langle x_1 + x_2 \\
&\qquad ,\lambda d.\ \mathbf{do}\ \langle\langle\rangle, dx\rangle \leftarrow \mathbf{scope}_{\mathcal{D}[a:\mathbb{R}]_2, \mathcal{D}[\mathbb{R}\times\mathbb{R}]_2}\ (\mathbf{do} \\
&\qquad\qquad\qquad (\mathbf{let}\ d' = d; d'' = \mathbf{onehot}_{\mathcal{D}[\mathbb{R}\times\mathbb{R}]_2, \overleftarrow{\times} *}\ \langle\rangle\ d'\ \mathbf{in} \\
&\qquad\qquad\qquad\quad \mathbf{accum}_{(b \in \mathcal{D}[a:\mathbb{R}, b:\mathbb{R}\times\mathbb{R}]_2), *}\ d'') \\
&\qquad\qquad\qquad (\mathbf{let}\ d' = d; d'' = \mathbf{onehot}_{\mathcal{D}[\mathbb{R}\times\mathbb{R}]_2, \overrightarrow{\times} *}\ \langle\rangle\ d'\ \mathbf{in} \\
&\qquad\qquad\qquad\quad \mathbf{accum}_{(b \in \mathcal{D}[a:\mathbb{R}, b:\mathbb{R}\times\mathbb{R}]_2), *}\ d'')) \\
&\qquad\qquad (\mathbf{let}\ d' = \mathbf{lfst}\ dx\ \mathbf{in}\ \mathbf{accum}_{(a \in \mathcal{D}[a:\mathbb{R}]_2), *}\ \langle\rangle\ d') \\
&\qquad\qquad (\mathbf{let}\ d' = \mathbf{lsnd}\ dx\ \mathbf{in}\ \mathbf{do}\ (\mathbf{let}\ d'' = m_2 \cdot d'\ \mathbf{in}\ \mathbf{return}\ \langle\rangle) \\
&\qquad\qquad\qquad\qquad\qquad (\mathbf{let}\ d'' = m_1 \cdot d'\ \mathbf{in}\ \mathbf{accum}_{(a \in \mathcal{D}[a:\mathbb{R}]_2), *}\ \langle\rangle\ d'')))\rangle
\end{aligned}$$

This term has no simplifications applied to it whatsoever, and yet it is already in first-order form. The primal is easily simplifiable by a compiler, and while the **accum**–**onehot** fusion described in Section 7.2 helps in the dual, further simplification there requires use of the properties of **scope** and **accum**, which are

[21] Were this version extended with support for lambda abstraction, source lambdas would naturally result in lambdas in the differentiated program. For "first-order", recall Section 2.1.1.

not local rewrite rules. Our implementation improves on this by avoiding introduction of effectful operations when unnecessary, so that fully exploiting **EVM**'s laws in optimisation of CHAD's output code is less critical; see Section 7.5.1.1.

7.4 Subset of the primal bindings

The primary goal of the split code transformation introduced in Section 7.3 was to make the output code first-order by construction for first-order input code. This improves the practicality of the algorithm and can significantly improve performance (assuming our compiler is not of the sufficiently smart variety).

A side-effect of the split transformation, however, is that we are now more explicit about which primal values are stored for the reverse pass: previously this was implicit in the construction of the backpropagator closure (namely: its free variables), but now we construct this tuple ourselves. This turns out to be simultaneously a good and a bad thing: on the one hand, we can be cleverer in which values we store and thereby reduce memory traffic as well as peak memory usage; on the other hand, we *must* be somewhat clever, lest the memory usage is actually worse than it was in Chapter 6. In fact, the storage discipline in Section 7.3 turns out to be naive enough that it makes things worse in practice.

7.4.1 The problem

The transformation collects primal bindings in $\mathcal{D}^0_\Gamma[t]$; this list gets promoted to an actual term at the top level (when it gets reified as actual let-bindings) and whenever control flow occurs (be it branching or iteration). With our source language, such control flow occurs for **case**, 'build' and 'fold'; in Figs. 7.5 to 7.7, the variables storing these bindings are named *tape* or *tapes*, as they fulfill a function similar to the tape in classical taping reverse AD. At these points, we indiscriminately store *all* entries contained in $\mathcal{D}^0_\Gamma[t]$, where t is the subterm in question, regardless of whether they are actually necessary for the reverse pass ($\mathcal{D}^2_{\Gamma,d}[t]$).

The problem is that for various scoping reasons, $\mathcal{D}^0_\Gamma[t]$ contains entries that are unused in $\mathcal{D}^2_{\Gamma,d}[t]$. In particular:

1. The variable r in $\overline{\mathcal{D}_\Gamma}[\mathbf{case}]$ is used only in the primal, and exists as a binding in $\mathcal{D}^0_\Gamma[\mathbf{case}]$ only because it is produced alongside *tape*, which *is* used in the reverse pass.

2. The variable a in $\overline{\mathcal{D}_\Gamma}[s\ !\ t]$ *is* in fact used in the reverse pass, but only rather trivially: we only need its length, not the entire array. Because our cost model assumes copying an object is free (i.e. let-binding and

$$\mathcal{D}[\Gamma]_1 \vdash_{\text{binds}} \mathcal{D}^0_\Gamma[t]$$
$$\mathcal{D}[\Gamma]_1, |\mathcal{D}^0_\Gamma[t]| \vdash \mathcal{D}^1_\Gamma[t] : \mathcal{D}[\tau]_1$$
$$\mathcal{D}[\Gamma]_1, |\mathcal{D}^0_\Gamma[t]|, d : \mathcal{D}[\tau]_2 \vdash \mathcal{D}^2_{\Gamma,d}[t] : \mathbf{EVM}\ \mathcal{D}[\Gamma]_2\ \mathbf{1}$$
$$\text{shorthand: } \overline{\mathcal{D}_\Gamma}[t] \coloneqq (\mathcal{D}^0_\Gamma[t], \mathcal{D}^1_\Gamma[t], \mathcal{D}^2_{\Gamma,d}[t])$$

$$\begin{aligned}
\overline{\mathcal{D}_\Gamma}[x : \tau] &= (\{\}, x : \mathcal{D}[\tau]_1, \mathbf{accum}_{(x:\mathcal{D}[\tau]_2 \in \mathcal{D}[\Gamma]_2),*}\ \langle\rangle\ d) \\
\overline{\mathcal{D}_\Gamma}[\mathbf{let}\ x : \tau = s\ \mathbf{in}\ t] &= (\mathcal{D}^0_\Gamma[s] +\!\!+ \{x = \mathcal{D}^1_\Gamma[s]\} +\!\!+ \mathcal{D}^0_{\Gamma,x:\tau}[t] \\
&\quad , \mathcal{D}^1_{\Gamma,x:\tau}[t], \mathbf{do}\ \langle\langle\rangle, dx\rangle \leftarrow \mathbf{scope}_{\mathcal{D}[\Gamma]_2,\mathcal{D}[\tau]_2}\ \mathcal{D}^2_{(\Gamma,x:\tau),d}[t] \\
&\qquad\qquad \mathcal{D}^2_{\Gamma,dx}[s]) \\
\overline{\mathcal{D}_\Gamma}[\langle\rangle] &= (\{\}, \langle\rangle, \mathbf{return}\ \langle\rangle) \\
\overline{\mathcal{D}_\Gamma}[\langle s, t\rangle] &= (\mathcal{D}^0_\Gamma[s] +\!\!+ \mathcal{D}^0_\Gamma[t] \\
&\quad , \langle \mathcal{D}^1_\Gamma[s], \mathcal{D}^1_\Gamma[t]\rangle, \mathbf{do}\ (\mathbf{let}\ d' = \mathbf{lfst}\ d\ \mathbf{in}\ \mathcal{D}^2_{\Gamma,d'}[s]) \\
&\qquad\qquad (\mathbf{let}\ d' = \mathbf{lsnd}\ d\ \mathbf{in}\ \mathcal{D}^2_{\Gamma,d'}[t])) \\
\overline{\mathcal{D}_\Gamma}[\text{fst}\ (t : \sigma \times \tau)] &= (\mathcal{D}^0_\Gamma[t], \text{fst}\ \mathcal{D}^1_\Gamma[t], \mathbf{let}\ d' = \mathbf{onehot}_{\mathcal{D}[\sigma\times\tau]_2,\overrightarrow{\times}*}\ \langle\rangle\ d\ \mathbf{in}\ \mathcal{D}^2_{\Gamma,d'}[t]) \\
\overline{\mathcal{D}_\Gamma}[\text{snd}\ (t : \sigma \times \tau)] &= (\mathcal{D}^0_\Gamma[t], \text{snd}\ \mathcal{D}^1_\Gamma[t], \mathbf{let}\ d' = \mathbf{onehot}_{\mathcal{D}[\sigma\times\tau]_2,\overrightarrow{\times}*}\ \langle\rangle\ d\ \mathbf{in}\ \mathcal{D}^2_{\Gamma,d'}[t]) \\
\overline{\mathcal{D}_\Gamma}\begin{bmatrix}\mathbf{case}\ s : \sigma \sqcup \tau\ \mathbf{of} \\ \quad \text{inl}\ x \to t_1 \\ \quad \text{inr}\ y \to t_2\end{bmatrix} &= (\mathcal{D}^0_\Gamma[s] +\!\!+ \\
&\qquad \{\langle r, \mathit{tape}\rangle = \mathbf{case}\ \mathcal{D}^1_\Gamma[s]\ \mathbf{of} \\
&\qquad\qquad \text{inl}\ x \to \mathbf{let}\ \mathcal{D}^0_{\Gamma,x:\sigma}[t_1] \\
&\qquad\qquad\qquad \mathbf{in}\ \langle \mathcal{D}^1_{\Gamma,x:\sigma}[t_1], \text{inl}\ \langle x, \overline{\mathcal{D}^0_{\Gamma,x:\sigma}[t_1]}\rangle\rangle \\
&\qquad\qquad \text{inr}\ y \to \mathbf{let}\ \mathcal{D}^0_{\Gamma,y:\tau}[t_2] \\
&\qquad\qquad\qquad \mathbf{in}\ \langle \mathcal{D}^1_{\Gamma,y:\tau}[t_2], \text{inr}\ \langle y, \overline{\mathcal{D}^0_{\Gamma,y:\tau}[t_2]}\rangle\rangle\} \\
&\quad , r \\
&\quad , \mathbf{do}\ d'' \leftarrow \mathbf{case}\ \mathit{tape}\ \mathbf{of} \\
&\qquad\qquad \text{inl}\ \langle x, \overline{\mathcal{D}^0_{\Gamma,x:\sigma}[t_1]}\rangle \to \mathbf{do} \\
&\qquad\qquad\quad \langle\langle\rangle, d'\rangle \leftarrow \mathbf{scope}_{\mathcal{D}[\Gamma]_2,\mathcal{D}[\sigma]_2}\ (\mathcal{D}^2_{(\Gamma,x:\sigma),d}[t_1]) \\
&\qquad\qquad\quad \mathbf{return}\ (\mathbf{linl}\ d') \\
&\qquad\qquad \text{inr}\ \langle y, \overline{\mathcal{D}^0_{\Gamma,y:\tau}[t_2]}\rangle \to \mathbf{do} \\
&\qquad\qquad\quad \langle\langle\rangle, d'\rangle \leftarrow \mathbf{scope}_{\mathcal{D}[\Gamma]_2,\mathcal{D}[\tau]_2}\ (\mathcal{D}^2_{(\Gamma,y:\tau),d}[t_2]) \\
&\qquad\qquad\quad \mathbf{return}\ (\mathbf{linr}\ d') \\
&\qquad \mathcal{D}^2_{\Gamma,d''}[s])
\end{aligned}$$

Figure 7.5: Full split transformation from Section 7.3 (1/3). Includes the changes from Section 7.2.

$$
\begin{aligned}
\overline{\mathcal{D}}_\Gamma[r] &= (\{\}, r, \mathbf{return}\ \langle\rangle) \\
\overline{\mathcal{D}}_\Gamma[\mathrm{sign}\ t] &= (\{\}, \mathrm{sign}\ t, \mathbf{return}\ \langle\rangle) \\
\overline{\mathcal{D}}_\Gamma[\mathrm{op}_\mathbb{R}(t_1, \ldots, t_n)] &= (\mathcal{D}^0_\Gamma[t_1] +\!\!+ \cdots +\!\!+ \mathcal{D}^0_\Gamma[t_n] +\!\!+ \{x_1 = \mathcal{D}^1_\Gamma[t_1], \ldots, x_n = \mathcal{D}^1_\Gamma[t_n]\} \\
&\quad , \mathrm{op}_\mathbb{R}(x_1, \ldots, x_n) \\
&\quad , \mathbf{do}\ (\mathbf{let}\ d' = \partial_1 \mathrm{op}_\mathbb{R}(d; x_1, \ldots, x_n)\ \mathbf{in}\ \mathcal{D}^2_{\Gamma,d'}[t_1]) \\
&\qquad\qquad \vdots \\
&\qquad\quad (\mathbf{let}\ d' = \partial_n \mathrm{op}_\mathbb{R}(d; x_1, \ldots, x_n)\ \mathbf{in}\ \mathcal{D}^2_{\Gamma,d'}[t_n])) \\
\overline{\mathcal{D}}_\Gamma[n] &= (\{\}, n, \mathbf{return}\ \langle\rangle) \\
\overline{\mathcal{D}}_\Gamma[\mathrm{op}_\mathbb{Z}(t_1, \ldots, t_n)] &= (\{\} \\
&\quad , \mathrm{op}_\mathbb{Z}(\mathbf{let}\ \mathcal{D}^0_\Gamma[t_1]\ \mathbf{in}\ \mathcal{D}^1_\Gamma[t_1], \ldots, \mathbf{let}\ \mathcal{D}^0_\Gamma[t_n]\ \mathbf{in}\ \mathcal{D}^1_\Gamma[t_n]) \\
&\quad , \mathbf{return}\ \langle\rangle) \\
\overline{\mathcal{D}}_\Gamma[\mathrm{build}\ s\ (i.\ t : \tau)] &= (\{n = \mathbf{let}\ \mathcal{D}^0_\Gamma[s]\ \mathbf{in}\ \mathcal{D}^1_\Gamma[s] \\
&\quad\ , \mathit{tapes} = \mathrm{build}\ n\ (i.\ \mathbf{let}\ \mathcal{D}^0_{\Gamma,i:\mathbb{Z}}[t]\ \mathbf{in}\ \overline{\mathcal{D}^0_{\Gamma,i:\mathbb{Z}}[t]})\} \\
&\quad , \mathrm{build}\ n\ (i.\ \mathbf{let}\ \overline{\mathcal{D}^0_{\Gamma,i:\mathbb{Z}}[t]} = \mathit{tapes}\ !\ i\ \mathbf{in}\ \mathcal{D}^1_{\Gamma,i:\mathbb{Z}}[t]) \\
&\quad , \mathbf{case}\ d\ \mathbf{of} \\
&\qquad \mathrm{inl}\ \langle\rangle \to \mathbf{return}\ \langle\rangle \\
&\qquad \mathrm{inr}\ da \to \mathbf{do} \\
&\qquad\quad \mathrm{sequence}\ (\mathrm{build}\ n\ (i.\ \mathbf{do}\ \mathbf{let}\ \underline{d' = da\ !\ i} \\
&\qquad\qquad\qquad\qquad\qquad\quad \mathbf{let}\ \mathcal{D}^0_{\Gamma,i:\mathbb{Z}}[t] = \mathit{tapes}\ !\ i \\
&\qquad\qquad\qquad\qquad\qquad\quad \mathbf{scope}_{\mathcal{D}[\Gamma]_2,\underline{\mathbf{1}}}\ (\mathcal{D}^2_{(\Gamma,i:\mathbb{Z}),d'}[t]) \\
&\qquad\qquad\qquad\qquad\qquad\quad \mathbf{return}\ \langle\rangle))) \\
&\qquad\quad \mathbf{return}\ \langle\rangle) \\
\overline{\mathcal{D}}_\Gamma[(s : \tau)\ !\ t] &= (\mathcal{D}^0_\Gamma[s] +\!\!+ \{a = \mathcal{D}^1_\Gamma[s], i = \mathbf{let}\ \mathcal{D}^0_\Gamma[t]\ \mathbf{in}\ \mathcal{D}^1_\Gamma[t]\} \\
&\quad , a\ !\ i, \mathbf{let}\ d' = \mathbf{onehot}_{\mathcal{D}[\tau]_2,!*}\ \langle i, \mathrm{length}\ a, \langle\rangle\rangle\ d\ \mathbf{in}\ \mathcal{D}^2_{\Gamma,d'}[s])
\end{aligned}
$$

Figure 7.6: Full split transformation from Section 7.3 (2/3).

$$
\begin{aligned}
&\overline{\mathcal{D}}_\Gamma[\text{fold } (p.\ s : \tau)\ t] = \\
&\quad (\mathcal{D}^0_\Gamma[t] +\!\!+ \{a = \mathcal{D}^1_\Gamma[t] \\
&\qquad\quad , \mathit{tree} = \text{fold } (p'.\ \textbf{let } \mathcal{D}^0_{\Gamma,p:\tau\times\tau}[s] \textbf{ in} \\
&\qquad\qquad\qquad\qquad \textbf{let } p = \langle \text{getA (fst } p'), \text{getA (snd } p')\rangle \\
&\qquad\qquad\qquad\qquad\quad\ y = \mathcal{D}^1_{\Gamma,p:\tau\times\tau}[s]\ \overline{\qquad\qquad} \\
&\qquad\qquad\qquad\qquad \textbf{in Node (fst } p')\ y\ (\mathcal{D}^0_{\Gamma,p:\tau\times\tau}[s])\ (\text{snd } p')) \\
&\qquad\qquad\qquad (\text{build (length } a)\ (i.\ \text{Leaf } i\ (a\ !\ i)))\} \\
&\quad , \text{getA } \mathit{tree} \\
&\quad , \textbf{do } \langle\langle\rangle, da\rangle \leftarrow \textbf{scope}_{\mathcal{D}[\Gamma]_2, (da : \text{Array } \mathcal{D}[\tau]_2)} \\
&\qquad\quad (\text{unTree } (\lambda p\ d'\ \mathit{tape}.\ \textbf{do} \\
&\qquad\qquad\qquad\quad \textbf{let } \overline{\mathcal{D}^0_{\Gamma,p:\tau\times\tau}[s]} = \mathit{tape} \\
&\qquad\qquad\qquad\quad \langle\langle\rangle, \langle d_1, d_2\rangle\rangle \leftarrow \textbf{scope}_{(\mathcal{D}[\Gamma]_2, da), \mathcal{D}[\tau\times\tau]_2}\ (\mathcal{D}^2_{(\Gamma,p:\tau\times\tau),d'}[s]) \\
&\qquad\qquad\qquad\quad \textbf{return } \langle d_1, d_2\rangle) \\
&\qquad\qquad\qquad d\ \mathit{tree}\ (\lambda i\ d'.\ \textbf{accum}_{da,!\,*}\ \langle i, \text{length } a, \langle\rangle\rangle\ d')) \\
&\qquad \textbf{let } d' = \text{inr } da \\
&\qquad \mathcal{D}^2_{\Gamma,d'}[t]\rangle)
\end{aligned}
$$

$$
\begin{aligned}
&\textbf{data Tree } a\ f = \text{Node (Tree } a\ f)\ a\ f\ (\text{Tree } a\ f) \mid \text{Leaf } \mathbb{Z}\ a \\
&\text{getA : Tree } a\ f \to a \\
&\text{getA (Node \_ } x \text{ \_ \_)} = x \\
&\text{getA (Leaf \_ } x) = x \\
&\text{unTree : Monad } m \Rightarrow (a \times a \to d \to f \to m\ (d \times d)) \to d \to \text{Tree } a\ f \\
&\qquad\qquad\qquad\qquad \to (\mathbb{Z} \to d \to m\ \mathbf{1}) \to m\ \mathbf{1} \\
&\text{unTree } g\ d\ (\text{Node } t_1\ \_\ f\ t_2)\ ac = \textbf{do } \langle d_1, d_2\rangle \leftarrow g\ \langle \text{getA } t_1, \text{getA } t_2\rangle\ d\ f \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \text{unTree } g\ d_1\ t_1\ ac \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \text{unTree } g\ d_2\ t_2\ ac \\
&\text{unTree } g\ d\ (\text{Leaf } i\ \_)\ ac = ac\ i\ d
\end{aligned}
$$

Figure 7.7: Full split transformation from Section 7.3 (3/3). Changes in unTree relative to Fig. 7.3 highlighted.

variable references are $O(1)$), storing the full argument array in $\mathcal{D}^0_\Gamma[s\ !\ t]$ is technically not a complexity problem, but in a (typical) implementation where structures of arrays are flattened, such copies may be expensive, and this storage of a is a complexity, memory usage *and* significant performance problem.

3. The same holds for a in $\overline{\mathcal{D}}_\Gamma[\text{fold}]$, although there is no complexity problem here as 'fold' itself takes (at least) linear time.

Although somewhat harder to spot in Figs. 7.5 to 7.7, there is one additional source of redundant stores for the dual that is not included in the list above. The problem is in the transformation meta-type, which currently prescribes that $\mathcal{D}^2_{\Gamma,d}[t]$ has $\mathcal{D}[\Gamma]_1$ as part of its environment:

$$\mathcal{D}[\Gamma]_1, |\mathcal{D}^0_\Gamma[t]|, d : \mathcal{D}[\tau]_2 \vdash \mathcal{D}^2_{\Gamma,d}[t] : \mathbf{EVM}\ \mathcal{D}[\Gamma]_2\ \mathbf{1}$$

However:

Observation. $\mathcal{D}[\Gamma]_1$ is always unused in $\mathcal{D}^2_{\Gamma,d}[t]$: all primal bindings used in the dual are already contained in $\mathcal{D}^0_\Gamma[t]$.

While this is not directly evident from the denotational semantics, it is easy to verify for our source language by inspection of Figs. 7.5 to 7.7; empirically, it further continues to hold for additional array primitives, as well as lambda abstraction and application. One way to explain this observation is to realise that dependence on $\mathcal{D}[\Gamma]_1$ in $\mathcal{D}^2_{\Gamma,d}[t]$ arises from a variable reference in the dual that points into $\mathcal{D}[\Gamma]_1$. As the bindings in Γ are not naturally directly available in the derivative rules (indeed, the rules are written to be parametrically polymorphic in Γ), such a variable reference should only arise from literal inclusion of a primal $\mathcal{D}^1_\Gamma[s]$ of some subterm s of t in t's dual. A primal is forward computation, so including a primal in the dual implies that some form of checkpointing (Section 2.2.9) is used; and indeed, addition of a checkpointing construct leads to selective reintroduction of the $\mathcal{D}[\Gamma]_1$ dependency in the dual (Section 7.5.1.5).

The advantage of dropping the $\mathcal{D}[\Gamma]_1$ dependency is naturally that we can drop some bindings:

4. As $\mathcal{D}[\Gamma]_1$ is unused in $\mathcal{D}^2_{\Gamma,d}[t]$, the binding x in $\mathcal{D}^0_\Gamma[\mathbf{let}]$ and x and y in *tape* in $\mathcal{D}^0_\Gamma[\mathbf{case}]$ are not needed for the reverse pass. For the same reason, i in $\mathcal{D}^2_{\Gamma,d}[\text{build}]$ and p in $\mathcal{D}^2_{\Gamma,d}[\text{fold}]$ need not be reintroduced in the dual.

Ensuring that we do not store redundant values for the reverse pass, especially when this happens many times because of an enclosing array combinator, has a significant impact on memory usage as well as performance.

7.4.2 Storing a subset of the primal bindings

To ensure that only a subset of the bindings in $\mathcal{D}^0_\Gamma[t]$ is stored by derivatives of terms such as **case** and 'build', we need to have a communication channel from the site where the bindings are generated (e.g. in $\mathcal{D}^0_\Gamma[\mathrm{op}_\mathbb{R}]$) to the site where they are consumed (e.g. $\overline{\mathcal{D}}_\Gamma[\mathbf{case}]$). We realise this communication channel as an additional output of the code transformation, written $\mathcal{D}^{0\mathrm{S}}_\Gamma[t]$:

$$
\begin{array}{l}
\mathcal{D}[\Gamma]_1 \vdash_{\text{binds}} \mathcal{D}^0_\Gamma[t] \\
\mathcal{D}^{0\mathrm{S}}_\Gamma[t] \subseteq |\mathcal{D}^0_\Gamma[t]| \qquad \text{(save list / subtape)} \\
\mathcal{D}[\Gamma]_1, |\mathcal{D}^0_\Gamma[t]| \vdash \mathcal{D}^1_\Gamma[t] : \mathcal{D}[\tau]_1 \\
\mathcal{D}^{0\mathrm{S}}_\Gamma[t], d : \mathcal{D}[\tau]_2 \vdash \mathcal{D}^2_{\Gamma,d}[t] : \mathbf{EVM}\ \mathcal{D}[\Gamma]_2\ \mathbf{1}
\end{array}
$$

Compared to Section 7.3, the save list is new, $\mathcal{D}^2_{\Gamma,d}[t]$ has lost $\mathcal{D}[\Gamma]_1$, and $|\mathcal{D}^0_\Gamma[t]|$ has been replaced with $\mathcal{D}^{0\mathrm{S}}_\Gamma[t]$ in the environment of $\mathcal{D}^2_{\Gamma,d}[t]$. The 'S' in the notation $\mathcal{D}^{0\mathrm{S}}_\Gamma[t]$ stands for "save list" or "subtape".

In terms of notation, we reuse the subset symbol $\subseteq$ to denote a sub*sequence* (order-preserving, not necessarily contiguous) of the right-hand side ($|\mathcal{D}^0_\Gamma[t]|$, i.e. the typing environment induced by the binding list $\mathcal{D}^0_\Gamma[t]$). Such a subsequence is a well-defined environment in itself because it contains name–type pairs only, not the bound terms, and our language is not dependently typed. Note that since subsequences are themselves lists, we have that $a \subseteq b$ and $c \subseteq d$ together imply $a +\!\!+ c \subseteq b +\!\!+ d$.

Updating the code transformation. For cases where all primal bindings are relevant for the dual, we simply copy the $\mathcal{D}^{0\mathrm{S}}$ of the subterms and add all new terms in $\mathcal{D}^0$. For example: (compare Figs. 7.5 to 7.7, pages 326 to 328)

$$
\begin{aligned}
&\mathcal{D}^{0\mathrm{S}}_\Gamma[x : \tau] = \{\} \\
&\mathcal{D}^{0\mathrm{S}}_\Gamma[\langle s, t\rangle] = \mathcal{D}^{0\mathrm{S}}_\Gamma[s] +\!\!+ \mathcal{D}^{0\mathrm{S}}_\Gamma[t] \\
&\mathcal{D}^{0\mathrm{S}}_\Gamma[\mathrm{op}_\mathbb{R}(t_1, \ldots, t_n)] = \mathcal{D}^{0\mathrm{S}}_\Gamma[t_1] +\!\!+ \cdots +\!\!+ \mathcal{D}^{0\mathrm{S}}_\Gamma[t_n] +\!\!+ \{x_1 : \tau_1, \ldots, x_n : \tau_n\}
\end{aligned}
$$

and analogously for $\langle\rangle$, fst, snd, r, sign, n, $\mathrm{op}_\mathbb{Z}$, build.

For $\mathrm{op}_\mathbb{R}$, a natural optimisation is to drop the x_i that are unnecessary for the partial derivatives of the operation in question; for example, linear operations like addition have partial derivatives independent of the input and hence can leave all x_i out of $\mathcal{D}^{0\mathrm{S}}_\Gamma[\mathrm{op}_\mathbb{R}]$. Concretely, we could have $\mathcal{D}^{0\mathrm{S}}_\Gamma[t_1 + t_2] = \mathcal{D}^{0\mathrm{S}}_\Gamma[t_1] +\!\!+ \mathcal{D}^{0\mathrm{S}}_\Gamma[t_2]$.

For the other term formers, we drop some bindings, possibly after cleverly introducing some new ones. For **let** and **case**, we simply drop the redundant bindings x and r: (compare Fig. 7.5):

$$
\begin{aligned}
&\mathcal{D}^{0\mathrm{S}}_\Gamma[\mathbf{let}\ x : \tau = s\ \mathbf{in}\ t] = \mathcal{D}^{0\mathrm{S}}_\Gamma[s] +\!\!+ \mathcal{D}^{0\mathrm{S}}_{\Gamma,x:\tau}[t] \\
&\mathcal{D}^{0\mathrm{S}}_\Gamma[\mathbf{case}\ s\ \mathbf{of}\ \{\ \mathrm{inl}\ x \to t_1 \mid \mathrm{inr}\ y \to t_2\ \}] = \mathcal{D}^{0\mathrm{S}}_\Gamma[s] +\!\!+ \{\mathit{tape}\}
\end{aligned}
$$

For (!) and 'fold', we need only 'length a' in the dual, not all of a, so we modify both $\mathcal{D}^0_\Gamma$ and $\mathcal{D}^{0\mathrm{S}}_\Gamma$:

$$\begin{aligned}
&\mathcal{D}^0_\Gamma[s\,!\,t] = \mathcal{D}^0_\Gamma[s] +\!\!+ \{a = \mathcal{D}^1_\Gamma[s], n = \text{length } a, i = \textbf{let } \mathcal{D}^0_\Gamma[t] \textbf{ in } \mathcal{D}^1_\Gamma[t]\}\\
&\mathcal{D}^{0\mathrm{S}}_\Gamma[s\,!\,t] = \mathcal{D}^{0\mathrm{S}}_\Gamma[s] +\!\!+ \{n, i\}\\
&\mathcal{D}^0_\Gamma[\text{fold } (p.\ s)\ t] = \mathcal{D}^0_\Gamma[t] +\!\!+ \{a = \mathcal{D}^1_\Gamma[t], n = \text{length } a, \mathit{tree} = \ldots\}\\
&\mathcal{D}^{0\mathrm{S}}_\Gamma[\text{fold } (p.\ s)\ t] = \mathcal{D}^{0\mathrm{S}}_\Gamma[t] +\!\!+ \{n, \mathit{tree}\}
\end{aligned}$$

Correspondingly, $\mathcal{D}^2_{\Gamma,d}[s\,!\,t]$ and $\mathcal{D}^2_{\Gamma,d}[\text{fold } (p.\ s)\ t]$ are modified to use n instead of 'length a'. This ensures that only the information that is actually needed in the dual is stored at control flow points.

To make use of the subtape we must finally modify $\overline{\mathcal{D}}_\Gamma[\textbf{case}]$, $\overline{\mathcal{D}}_\Gamma[\text{build}]$ and $\overline{\mathcal{D}}_\Gamma[\text{fold}]$ to use $\mathcal{D}^{0\mathrm{S}}$ instead of the plain list of primal bindings; the result is given in Figs. 7.8 and 7.9. Changes compared to Figs. 7.5 to 7.7 are highlighted. Notice that additionally, the preservation of x and y is removed in $\overline{\mathcal{D}}_\Gamma[\textbf{case}]$ and the reintroduction of p is removed in $\mathcal{D}^2_{\Gamma,d}[\text{fold}]$ (although 'unTree' is left as-is to reduce churn). In $\overline{\mathcal{D}}_\Gamma[\text{build}]$, we had to bundle the construction of *tapes* and the primal x to ensure that redundant bindings in $\mathcal{D}^0_{\Gamma,i:\mathbb{Z}}[t]$ do not get persisted in an array; furthermore, as i is no longer needed for $\mathcal{D}^2_{(\Gamma,i:\mathbb{Z}),d'}[t]$, the 'build' in the dual can revert to a less-general 'zipWith'.

7.4.3 Example

The example discussed in Section 7.3.3 (originally from Section 5.1.5, pages 218 to 222) does not change after applying the improvements from this section, as there is no control flow or iteration construct to make use of reduced store lists. As soon as we introduce such a construct, however, a change can be observed. Consider:

$$\begin{aligned}
&k : \mathbb{Z}, a : \text{Array } \mathbb{R} \vdash t_2 : \text{Array } \mathbb{R}\\
&t_2 = \text{build } k\ (i.\ \textbf{let } z = a\,!\,i\\
&\qquad\qquad\qquad\qquad \textbf{in } z + 2 \cdot z)
\end{aligned}$$

$$\mathcal{D}[\Gamma]_1 \vdash_{\text{binds}} \mathcal{D}^0_\Gamma[t]$$
$$\mathcal{D}^{0S}_\Gamma[t] \subseteq |\mathcal{D}^0_\Gamma[t]|$$
$$\mathcal{D}[\Gamma]_1, |\mathcal{D}^0_\Gamma[t]| \vdash \mathcal{D}^1_\Gamma[t] : \mathcal{D}[\tau]_1$$
$$\mathcal{D}^{0S}_\Gamma[t], d : \mathcal{D}[\tau]_2 \vdash \mathcal{D}^2_{\Gamma,d}[t] : \mathbf{EVM}\ \mathcal{D}[\Gamma]_2\ \mathbf{1}$$
shorthand: $\overline{\mathcal{D}}_\Gamma[t] \coloneqq (\mathcal{D}^0_\Gamma[t], \overline{\mathcal{D}^{0S}_\Gamma[t]}, \mathcal{D}^1_\Gamma[t], \mathcal{D}^2_{\Gamma,d}[t])$

$$\overline{\mathcal{D}}_\Gamma\begin{bmatrix}\mathbf{case}\ s : \sigma \sqcup \tau\ \mathbf{of}\\ \quad \text{inl}\ x \to t_1\\ \quad \text{inr}\ y \to t_2\end{bmatrix} = (\mathcal{D}^0_\Gamma[s] +\!\!+$$
$$\quad \{\langle r, tape\rangle = \mathbf{case}\ \mathcal{D}^1_\Gamma[s]\ \mathbf{of}$$
$$\quad\quad \text{inl}\ x \to \mathbf{let}\ \mathcal{D}^0_{\Gamma,x:\sigma}[t_1]$$
$$\quad\quad\quad \mathbf{in}\ \langle \mathcal{D}^1_{\Gamma,x:\sigma}[t_1], \text{inl}\ (\overline{\mathcal{D}^{0S}_{\Gamma,x:\sigma}[t_1]})\rangle$$
$$\quad\quad \text{inr}\ y \to \mathbf{let}\ \mathcal{D}^0_{\Gamma,y:\tau}[t_2]$$
$$\quad\quad\quad \mathbf{in}\ \langle \mathcal{D}^1_{\Gamma,y:\tau}[t_2], \text{inr}\ (\overline{\mathcal{D}^{0S}_{\Gamma,y:\tau}[t_2]})\rangle\}$$
$$, \mathcal{D}^{0S}_\Gamma[s] +\!\!+ \{tape\}$$
$$, r$$
$$, \mathbf{do}\ d'' \leftarrow \mathbf{case}\ tape\ \mathbf{of}$$
$$\quad\quad \text{inl}\ (\overline{\mathcal{D}^{0S}_{\Gamma,x:\sigma}[t_1]}) \to \mathbf{do}$$
$$\quad\quad\quad \langle\langle\rangle, d'\rangle \leftarrow \mathbf{scope}_{\mathcal{D}[\Gamma]_2, \mathcal{D}[\sigma]_2}\ (\mathcal{D}^2_{(\Gamma,x:\sigma),d}[t_1])$$
$$\quad\quad\quad \mathbf{return}\ (\mathbf{linl}\ d')$$
$$\quad\quad \text{inr}\ (\overline{\mathcal{D}^{0S}_{\Gamma,y:\tau}[t_2]}) \to \mathbf{do}$$
$$\quad\quad\quad \langle\langle\rangle, d'\rangle \leftarrow \mathbf{scope}_{\mathcal{D}[\Gamma]_2, \mathcal{D}[\tau]_2}\ (\mathcal{D}^2_{(\Gamma,y:\tau),d}[t_2])$$
$$\quad\quad\quad \mathbf{return}\ (\mathbf{linr}\ d')$$
$$\quad \mathcal{D}^2_{\Gamma,d''}[s])$$

$$\overline{\mathcal{D}}_\Gamma[\text{build}\ s\ (i.\ t : \tau)] = (\{n = \mathbf{let}\ \mathcal{D}^0_\Gamma[s]\ \mathbf{in}\ \mathcal{D}^1_\Gamma[s]$$
$$\quad , \langle x, tapes\rangle = \text{unzip}\ (\text{build}\ n\ (i.\ \mathbf{let}\ \mathcal{D}^0_{\Gamma,i:\mathbb{Z}}[t]$$
$$\quad\quad\quad\quad\quad\quad\quad\quad \mathbf{in}\ \langle \mathcal{D}^1_{\Gamma,i:\mathbb{Z}}[t], \overline{\mathcal{D}^{0S}_{\Gamma,i:\mathbb{Z}}[t]}\rangle))\}$$
$$, \{n, tapes\}$$
$$, x$$
$$, \mathbf{case}\ d\ \mathbf{of}$$
$$\quad \text{inl}\ \langle\rangle \to \mathbf{return}\ \langle\rangle$$
$$\quad \text{inr}\ da \to \mathbf{do}$$
$$\quad\quad \text{sequence}\ (\text{zipWith}\ (d'\ (\overline{\mathcal{D}^{0S}_{\Gamma,i:\mathbb{Z}}[t]}).\ \mathbf{do}$$
$$\quad\quad\quad\quad\quad \mathbf{scope}_{\mathcal{D}[\Gamma]_2, \underline{\mathbf{1}}}\ (\mathcal{D}^2_{(\Gamma,i:\mathbb{Z}),d'}[t])$$
$$\quad\quad\quad\quad\quad \mathbf{return}\ \langle\rangle)$$
$$\quad\quad\quad\quad da\ tapes)$$

Figure 7.8: Transformation for **case** and 'build' updated to make use of $\mathcal{D}^{0S}$. Changes with respect to Figs. 7.5 and 7.6 highlighted. Continued in Fig. 7.9.

$$
\begin{aligned}
&\mathcal{D}_\Gamma[\text{fold } (p.\ s)\ (t : \text{Array } \tau)] = \\
&\quad (\mathcal{D}^0_\Gamma[t] +\!\!+ \{a = \mathcal{D}^1_\Gamma[t] \\
&\qquad\quad , tree = \text{fold } (p'.\ \textbf{let } \mathcal{D}^0_{\Gamma,p:\tau\times\tau}[s] \textbf{ in} \\
&\qquad\qquad\qquad\qquad \textbf{let } p = \langle \text{getA } (\text{fst } p'), \text{getA } (\text{snd } p')\rangle \\
&\qquad\qquad\qquad\qquad\quad\ y = \mathcal{D}^1_{\Gamma,p:\tau\times\tau}[s] \\
&\qquad\qquad\qquad\qquad \textbf{in } \text{Node } (\text{fst } p')\ y\ (\overline{\mathcal{D}^{0S}_{\Gamma,p:\tau\times\tau}[s]})\ (\text{snd } p')) \\
&\qquad\qquad\qquad (\text{build } (\text{length } a)\ (i.\ \text{Leaf } i\ (a\ !\ i)))\} \\
&\quad , \text{getA } tree \\
&\quad , \textbf{do } \langle\langle\rangle, da\rangle \leftarrow \textbf{scope}_{\mathcal{D}[\Gamma]_2,(da:\text{Array } \mathcal{D}[\tau]_2)} \\
&\qquad\quad (\text{unTree } (\lambda\_\ d'\ tape.\ \textbf{do} \\
&\qquad\qquad\qquad \textbf{let } \overline{\mathcal{D}^{0S}_{\Gamma,p:\tau\times\tau}[s]} = tape \\
&\qquad\qquad\qquad \langle\langle\rangle, \langle d_1, d_2\rangle\rangle \leftarrow \textbf{scope}_{(\mathcal{D}[\Gamma]_2,da),\mathcal{D}[\tau\times\tau]_2}\ (\mathcal{D}^2_{(\Gamma,p:\tau\times\tau),d'}[s]) \\
&\qquad\qquad\qquad \textbf{return } \langle d_1, d_2\rangle) \\
&\qquad\qquad\quad d\ tree\ (\lambda i\ d'.\ \textbf{accum}_{da,!*}\ \langle i, \text{length } a, \langle\rangle\rangle\ d')) \\
&\qquad \textbf{let } d' = \text{inr } da \\
&\qquad \mathcal{D}^2_{\Gamma,d'}[t]\rangle)
\end{aligned}
$$

Figure 7.9: Transformation for 'fold' updated to make use of $\mathcal{D}^{0S}$. Changes with respect to Fig. 7.7 highlighted. Belongs with Fig. 7.8.

Assembling its derivative as $\textbf{let } \mathcal{D}^0_{k,a}[t_2] \textbf{ in } \langle \mathcal{D}^1_{k,a}[t_2], \lambda d.\ \mathcal{D}^2_{(k,a),d}[t_2]\rangle$, we get after some simplification:

$$
\begin{aligned}
&\textbf{let } \langle x, tapes\rangle = \text{unzip } (\text{build } k\ (i.\ \textbf{let } z = a\ !\ i \textbf{ in } \langle z + 2\cdot z, \langle \text{length } a, i, 2, z\rangle\rangle)) \\
&\textbf{in } \langle x, \lambda d.\ \textbf{case } d \textbf{ of} \\
&\qquad\quad \text{inl } \langle\rangle \rightarrow \textbf{return } \langle\rangle \\
&\qquad\quad \text{inr } da \rightarrow \textbf{do} \\
&\qquad\qquad \text{sequence } (\text{zipWith } (d'\ \langle n, i, x_1, x_2\rangle.\ \textbf{do} \\
&\qquad\qquad\qquad\qquad \textbf{scope}_{(k,a),\mathbf{1}}\ (\textbf{do } \langle\langle\rangle, dz\rangle \leftarrow \textbf{scope}_{(k,a,i),\mathbb{R}}\ (\textbf{do} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{accum}_{(z\in k,a,i,z),*}\ \langle\rangle\ d' \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{accum}_{(z\in k,a,i,z),*}\ \langle\rangle\ (x_1\cdot d')) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{accum}_{(a\in k,a,i),!*}\ \langle i, n, \langle\rangle\rangle\ dz) \\
&\qquad\qquad\qquad\qquad \textbf{return } \langle\rangle) \\
&\qquad\qquad\qquad\quad da\ tapes) \\
&\qquad\qquad \textbf{return } \langle\rangle\rangle
\end{aligned}
$$

The stores of z (arising from the let-binding) and a (from the indexing) in *tapes* are successfully avoided. However, indexing still stores the length of the array (length a) and the index (i) for use in its **accum** statement in the reverse pass, and multiplication still stores both its arguments (2 and z) in order to compute its partial derivatives. All of these are in principle avoidable:

1. 'length a' can be bound once outside of the 'build' and need not be saved k times;

2. 'i' can be available in the reverse pass if the 'zipWith' is reverted to a 'build' again;

3. '2' is clearly known statically and need not be saved at runtime; and

4. The z is only used to compute the partial derivative of multiplication with respect to the expression '2', which ends up unused.

Furthermore, even if the partial derivative of multiplication with respect to its first argument were required, recomputing $a\,!\,i$ would probably be better than storing z here. Our implementation (Section 7.5) is able to avoid some of these:

1. No dynamic sparsity is used for array accumulators, meaning that 'length a' need not be saved (Section 7.5.1.2);

4. A general-purpose usage analysis after AD with product-element granularity is able to remove the unused z store in *tapes* (Section 7.5.1.3).

A satisfactory solution for 'i' and '2' is, however, future work (see Section 7.6).

7.5 Implementation

We have an implementation[22] of the CHAD algorithm that includes the improvements in this chapter, as well as additional optimisations that benefit practical runtime. An overview of the most important features is given in Section 7.5.1; more thorough justification and discussion, handling of the remaining efficiency problems, and benchmarking are left to future work.

The implementation is written in Haskell and uses a well-typed and well-scoped, intrinsically typed De Bruijn AST for representing the embedded program. This means that ill-typed programs cannot be represented, thus showing the type-correctness (albeit not the semantical correctness) of the AD algorithm. Usage of such an strictly typed AST in compilers is uncommon; an example of prior use in a practical system is in the Accelerate compiler [Chakravarty et al. 2011].

Instead of the monadic representation used so far for the (local) accumulation effect (i.e. **EVM**), the implementation uses a syntax based on effects and handlers. This reduces the number of term formers in the syntax that have no runtime equivalent (such as 'sequence' and '**return**', as accumulation is actually implemented as mutation, not a functional state monad) and makes it more obvious that the effect is commutative.

While our implementation generates C code in the backend for efficient execution, this C code is sequential, and no loop fusion (Section 2.1.4) is performed

[22] `https://git.tomsmeding.com/chad-fast`

prior to code generation. This means that performance is acceptable but not yet comparable to the state of the art, even ignoring the remaining infelicities in the AD algorithm.

The input language supported by the implementation is the larger language in Section 7.1, i.e. including nested arrays and coproduct types but excluding lambda functions; some additional array combinators are also supported.

7.5.1 Additional optimisations

We briefly describe the major features of the implementation that are not adequately described elsewhere in this thesis.

7.5.1.1 Selective accumulation

Accumulators are beneficial to prevent one-hots from creating complexity issues, but they also result in overhead, as they force cotangent accumulation to happen in memory instead of in registers. This also makes the produced code harder to optimise for a compiler, as one needs to make non-trivial use of laws for **EVM**'s methods. Thus, to lessen the impact of accumulators, we allow the transformation to choose to selectively pass some cotangents naively, as in Chapter 5.

Specifically, the environment Γ is split disjointly into two environments Γ_A and Γ_M: the accumulation environment and the merge environment.[23] This splitting is considered an additional input to the code transformation, manifesting in an updated index to $\mathcal{D}^2$. At binding sites (**let**, **case**, etc.), newly bound variables can be included in either Γ_A or Γ_M at will — the choice determines how cotangents for the variable are returned from $\mathcal{D}^2_{\Gamma_A,\Gamma_M,d}[t]$ below, and matters for performance and (occasionally) complexity but has no impact on semantics.

Using this split environment, the meta-type of the transformation changes as follows: (differences with Section 7.4 highlighted)

$$\begin{aligned}
&\mathcal{D}[\Gamma]_1 \vdash_{\text{binds}} \mathcal{D}^0_\Gamma[t] \\
&\mathcal{D}^{0\mathrm{S}}_\Gamma[t] \subseteq |\mathcal{D}^0_\Gamma[t]| \\
&\mathcal{D}[\Gamma]_1, |\mathcal{D}^0_\Gamma[t]| \vdash \mathcal{D}^1_\Gamma[t] : \mathcal{D}[\tau]_1 \\
&\mathcal{D}^{0\mathrm{S}}_\Gamma[t], d : \mathcal{D}[\tau]_2 \vdash \mathcal{D}^2_{\Gamma_A,\Gamma_M,d}[t] : \mathbf{EVM}\ \mathcal{D}[\Gamma_A]_2\ \mathrm{Tup}[\mathcal{D}[\Gamma_M]_2]
\end{aligned}$$

where $\mathrm{Tup}[x_1 : \tau_1, \ldots, x_n : \tau_n] \coloneqq \tau_1 \times \cdots \times \tau_n$.

At array combinators such as 'build', care must be taken that free variables are in Γ_A, not Γ_M, so that indexing inside the loop has an efficient derivative. This can be accomplished by opening up a number of **scope** clauses around such an array combinator and promoting variables from Γ_M to Γ_A using those.

[23] So called because the $\mathcal{D}[\Gamma_M]_2$ tuples from adjacent subterms, possibly with different free variables, get "merged" upwards. "Add" does not work as 'A' is taken.

The packing and unpacking of $\mathcal{D}[\Gamma_M]_2$ tuples results in an additional $O(\#\Gamma_M)$ factor in the time complexity, but assuming standard compiler optimisations, this additional cost generally only persists at dynamic control flow sites, i.e. points where the source program branches. In straight-line code, construction and elimination of the $\mathcal{D}[\Gamma_M]_2$ tuples cancels. This optimisation significantly simplifies the output code of CHAD.

7.5.1.2 Mixed static-dynamic sparsity

Since Chapter 6, we add dynamic sparsity wrappers ($\mathbf{1}\sqcup$) around every non-trivial type for the sake of cheap zero and plus in the event that some cotangents, or parts of cotangents, are zero. However, dynamic sparsity wrappers are anathema for performance: not only do they introduce runtime branches, they also separate the computation into various small pieces separated by dynamic control flow, which incapacitates various compiler optimisations. Wrappers around intermediate values in a straight-line computation can generally be eliminated by beta-reduction, but once values are shared this already becomes trickier.

For performance, it is essential that these unnecessary dynamic wrappers be elided. Fortunately, it turns out that in the majority of situations, it is possible to statically conclude for most cotangents that it is either always going to be zero, or always going to be non-zero[24]. In fact, even a single-pass sparsity propagation approach running simultaneously with differentiation can detect the vast majority of such unnecessary dynamic sparsity. The result that we obtain is that always-absent cotangents are replaced with nothing at all (i.e. $\mathbf{1}$), and always-present cotangents are represented (partially) densely.

Controlled cotangent sparsification. The implementation of this idea is somewhat involved in the details, but the high-level design is simple. We start by reverting to fully dense cotangents:[25]

$$\mathcal{D}[\mathbb{R}]_2 = \mathbb{R} \qquad \mathcal{D}[\mathbb{Z}]_2 = \mathbf{1} \qquad \mathcal{D}[\mathbf{1}]_2 = \mathbf{1}$$
$$\mathcal{D}[\sigma \times \tau]_2 = \mathcal{D}[\sigma]_2 \times \mathcal{D}[\tau]_2 \qquad \mathcal{D}[\sigma \sqcup \tau]_2 = \mathbf{1} \sqcup \mathcal{D}[\sigma]_2 \sqcup \mathcal{D}[\tau]_2$$
$$\mathcal{D}[\text{Array } \tau]_2 = \text{Array } \mathcal{D}[\tau]_2$$

[24] A cotangent produced by non-trivial computation counts as "non-zero" here, even if it could just so happen to contain only `0.0` values in practice.

[25] We retain a well-defined zero for $\mathcal{D}[\sigma \sqcup \tau]_2$ here.

Afterwards, we reintroduce sparsity in a controlled fashion by mixing static and dynamic sparsity. Define a grammar of sparsifications, or sparsity descriptions:[26]

$$\frac{}{A : \mathbf{1} \le \tau} \qquad \frac{\mathfrak{s} : \tau' \le \tau}{S\,\mathfrak{s} : \mathbf{1} \sqcup \tau' \le \tau}$$

$$\frac{}{R : \mathbb{R} \le \mathbb{R}} \qquad \frac{\mathfrak{s}_1 : \sigma' \le \sigma \quad \mathfrak{s}_2 : \tau' \le \tau}{P\,\mathfrak{s}_1\,\mathfrak{s}_2 : \sigma' \times \tau' \le \sigma \times \tau} \qquad \frac{\mathfrak{s}_1 : \sigma' \le \sigma \quad \mathfrak{s}_2 : \tau' \le \tau}{C\,\mathfrak{s}_1\,\mathfrak{s}_2 : \mathbf{1} \sqcup \sigma' \sqcup \tau' \le \mathbf{1} \sqcup \sigma \sqcup \tau}$$

$$\frac{\mathfrak{s} : \tau' \le \tau}{T\,\mathfrak{s} : \mathrm{Array}\ \tau' \le \mathrm{Array}\ \tau}$$

In the judgement $\mathfrak{s} : \sigma \le \tau$, $\mathfrak{s}$ describes how σ is a sparsified version of τ. The productions R (real), P (product), C (coproduct) and T (tensor[27]) simply recurse over the grammar of types; A (absent) introduces static sparsity, and S (sparse) introduces dynamic sparsity. For example, for any τ we have:

$$P\ (S\ (T\ R))\ A : (\mathbf{1} \sqcup \mathrm{Array}\ \mathbb{R}) \times \mathbf{1} \le \mathrm{Array}\ \mathbb{R} \times \tau$$

which might describe the actual representation of a cotangent of type $\mathrm{Array}\ \mathbb{R} \times \tau$ for some τ, where the second component is *statically* known to be zero, whereas the array may or may not be present depending on yet unknown runtime values. We assume that $\mathbf{1}$ has no runtime representation after compilation.

As written, the induced relation $\le$ on types is not quite a partial order, because $A : \mathbf{1} \le \mathbf{1} \sqcup \mathbf{1}$ and $S\,A : \mathbf{1} \sqcup \mathbf{1} \le \mathbf{1}$ violate antisymmetry — the fact that "$S\,A$" is never helpful notwithstanding. If the $\mathbf{1}$ introduced by 'A' would be different from the normal unit type, then given a fixed τ, the set of types τ' for which $\mathfrak{s} : \tau' \le \tau$ is derivable becomes a join-semilattice under $\le$ (i.e. a partial order with least upper bounds). However, the resulting join operation is not the desired one for combining two cotangent values.[28]

Updating the meta-type. Let $\mathfrak{s} : \delta \le \mathcal{D}[\tau]_2$ be an additional parameter of the code transformation, where τ is the return type of the input term t and δ is fixed by $\mathfrak{s}$ and τ. Working from the version in Section 7.5.1.1, we update the highlighted parts:

$$\begin{aligned}
&\mathcal{D}[\Gamma]_1 \vdash_{\text{binds}} \mathcal{D}^0_\Gamma[t] \\
&\mathcal{D}^{0\mathrm{S}}_\Gamma[t] \subseteq |\mathcal{D}^0_\Gamma[t]| \\
&\mathcal{D}[\Gamma]_1, |\mathcal{D}^0_\Gamma[t]| \vdash \mathcal{D}^1_\Gamma[t] : \mathcal{D}[\tau]_1 \\
&\mathcal{D}^{2\mathrm{s}}_{\Gamma_M}[t] : \delta'_M \le \mathcal{D}[\Gamma_M]_2 \\
&\mathcal{D}^{0\mathrm{S}}_\Gamma[t], d : \delta \vdash \mathcal{D}^2_{\Gamma_A,\Gamma_M,d,\mathfrak{s}}[t] : \mathbf{EVM}\ \mathcal{D}[\Gamma_A]_2\ \delta'_M
\end{aligned}$$

[26] These rules describe `data Sparse` in `CHAD.AST.Sparse.Types` in the implementation.

[27] The A is already taken.

[28] $T\,A \vee S\ (T\ (T\,\mathfrak{s})) = T\ (T\,\mathfrak{s})$, but values with these two sparsities cannot be combined to a $T\ (T\,\mathfrak{s})$ because the size of the inner arrays may not be known. Nonhomogeneous addition (see later) would return $T\ (S\ (T\,\mathfrak{s}))$ here, which is greater than neither input in the relation.

Thus, $\mathfrak{s}$ describes the sparsity of the incoming cotangent to this term t (i.e. the argument to its backpropagator in naive CHAD), and $\mathcal{D}^{2s}_{\Gamma_M}[t]$ describes the sparsity of the direct "merge-style" cotangents (as in Section 7.5.1.1) passed up to the parent term (i.e. some of the outputs of that naive backpropagator). The initial $\mathfrak{s}$ at the top level is set to the sparsity of the top-level cotangent (probably dense), and sparsity descriptions are passed on in reverse control flow direction in the only natural way, thus implementing the mentioned single-pass sparsity propagation.

Notable is that only these merge-style cotangents are sparsified; applying static sparsity also to accumulators is possible but unimplemented, as it would require a two-pass algorithm that first determines the union of all accumulation sparsities for a particular accumulator, and afterwards generates the appropriate code at all accumulation sites.[29] Instead, the implementation always initialises accumulators with dense zero cotangents, and attempts are made to reduce the number of such accumulators (see Section 7.5.1.4 below).

Nonhomogeneous addition. With the concrete representation of cotangents now dependent on the term they are produced by, having just homogeneous addition of monoids ($\tau \to \tau \to \tau$ for monoids τ) is insufficient: we may need to add cotangents with differing static sparsity structures, the result being a union of both structures, not unlike Fig. 6.7 on page 249. We can attempt to implement this nonhomogeneous addition by writing the following function in the compiler: (for a monoid type τ)

$$\begin{aligned}\text{generatePlus} : \forall \tau\, \tau_1\, \tau_2.\ (\mathfrak{s}_1 : \tau_1 \le \tau) \to (\mathfrak{s}_2 : \tau_2 \le \tau)\\ \to \exists \tau_3.\ ((\mathfrak{s}_3 : \tau_3 \le \tau) \times \text{Term}\ (\tau_1 \to \tau_2 \to \tau_3))\end{aligned}$$

writing 'Term' for the type of ASTs. Here, τ_3 and $\mathfrak{s}_3$ are produced at compile time but the addition itself is performed at runtime. Note that τ_3 could be, but is not necessarily, the join of τ_1 and τ_2 in the partial order suspended below τ, as sometimes a less-sparse option is more efficient (e.g. if one of τ_1, τ_2 has a cheap zero) or conversely the theoretical join cannot be practically realised due to unknown array lengths.

Unfortunately, 'generatePlus' is insufficient: whenever branching occurs, a *choice* needs to be made between two cotangents from different sources; thus, we need to be able to make two cotangents with different sparsity structures compatible without actually summing them. We can do this by giving two more arguments to 'generatePlus': booleans indicating whether the first, respectively the second, argument of the addition will certainly be present at runtime or

[29] With `SplitLets` mentioned in Section 7.5.1.4, applying static sparsity to accumulators is fortunately less crucial.

not.[30] Potential absence of an argument can result in additional 'S' nodes in the generated sparsity $\mathfrak{s}_3$ to avoid having to initialise large dense zeros. This function is implemented as `sparsePlusS` in `CHAD.AST.Sparse` in the implementation.

With this, the transformation can be updated. Homogeneous addition is occasionally required when an array of cotangents must be summed, but in those cases the representations were already homogeneous beforehand, and homogeneous addition suffices.

7.5.1.3 Store pruning

In Section 7.4.3, we noted that despite some efforts to the contrary, more primals are stored for the dual than would be required or ideal. One way to reduce the severity of this problem is to remove stores that are clearly unused in the dual; this can be done using a compiler optimisation independent of AD. Specifically, we implemented a usage analysis, or liveness analysis, that tracks usage not only of variables but also of components in product types and that can thus detect that certain "tape" elements are in fact unused and elide them. Our implementation does this in `pruneExpr` in `CHAD.AST.Count`.

This analysis works well on straight-line code, but dynamic control flow can prevent propagation of usage information and thus foil the analysis. Addressing this problem not via an AD-agnostic optimisation pass after AD but inside the AD algorithm, building on the literature in taping AD (e.g. the to-be-recorded analysis of Hascoët et al. [2005]), has the potential to be more powerful. See also 'Primal storage without duplication' in Section 7.6.

7.5.1.4 Accumulator reuse

Consider the following source program:

$$\begin{aligned}&\textbf{let } a = \text{build } n\ (i.\ \ldots)\\&\textbf{in let } b = a\\&\quad\textbf{in } \text{build } k\ (i.\ a\,!\,i + b\,!\,i)\end{aligned}$$

Suppose that b is not optimised away before AD. Then both a and b should be put in Γ_A by the let-bindings (recall Section 7.5.1.1) to avoid pessimising the dual of indexing. As a result, two accumulators are allocated and initialised; upon closure of the **scope** for b, its collected cotangent is added to the cotangent for a by bulk-accumulating it (using $\mathbf{accum}_{\text{Array}\,\mathbb{R},*}$). This is suboptimal, as ideally we would allocate only one zero and accumulate into each of its fields only once, thus

[30] Note that even if both arguments are always present in an addition, this may not hold any more for subvalues: when adding $S\ (T\ \mathfrak{s}_1)$ and $T\ \mathfrak{s}_2$ we want to recurse to adding $T\ \mathfrak{s}_1$ and $T\ \mathfrak{s}_2$, but here the first argument may not be present any more.

resulting in n zeros and k additions instead of the $2n$ zeros and $2k + n$ additions that the above code naively generates.

In simple cases, our implementation avoids this inefficiency by reusing an existing accumulator if one would be allocated for the same original program value. A data flow analysis first attempts to determine which program variables (with product component granularity) provably refer to the same value at runtime (`CHAD.Analysis.Identity`). Afterwards, in the main AD transformation, runtime value identities (as determined by the analysis) are matched with accumulators in scope for them; if a new accumulator would be created for an identity that already has an open accumulator, that accumulator is reused through an environment substitution: essentially, the inlining of b in the above example is done virtually at differentiation time.

In order to work even if the array in question is part of a larger product, let-bindings of product type are split into let-bindings of the individual components before AD (`CHAD.AST.SplitLets`), so that each contained array gets its own binding.

7.5.1.5 Checkpointing

A very simple addition to the language with an equally simple differentiation rule allows a disproportionately effective form of checkpointing in CHAD. In the form of Chapter 6, we have:

$$\begin{aligned} t &::= \cdots \mid \text{recompute } t \\ \mathcal{D}_\Gamma[\text{recompute } t] &= \langle \text{fst } \mathcal{D}_\Gamma[t], \text{snd } \mathcal{D}_\Gamma[t] \rangle \end{aligned} \tag{7.5}$$

This is all. Note that this definition intentionally duplicates code, resulting in rather different operational behaviour from $\mathcal{D}_\Gamma[\text{recompute } t] = \mathcal{D}_\Gamma[t]$!

The recomputation becomes much more explicit if we translate to the latest version of the code transformation (Section 7.5.1.2):

$$\begin{aligned} \mathcal{D}^0_\Gamma[\text{recompute } t] &= \{x_1 : \mathcal{D}[\tau_1]_1 = x_1, \ldots, x_n : \mathcal{D}[\tau_n]_1 = x_n\} \\ \mathcal{D}^{0\mathbb{S}}_\Gamma[\text{recompute } t] &= \{FV(t)\} && (\text{for } x_i : \tau_i \in FV(t)) \\ \mathcal{D}^1_\Gamma[\text{recompute } t] &= \mathcal{D}^1_\Gamma[t] \\ \mathcal{D}^{2s}_\Gamma[\text{recompute } t] &= \mathcal{D}^{2s}_\Gamma[t] \\ \mathcal{D}^2_{\Gamma,d,\mathbb{s}}[\text{recompute } t] &= \textbf{let } \mathcal{D}^0_\Gamma[t] \textbf{ in } \mathcal{D}^2_{\Gamma,d,\mathbb{s}}[t] \end{aligned}$$

The recomputation aspect is captured by the inclusion of the primal bindings $\mathcal{D}^0_\Gamma[t]$ in $\mathcal{D}^2_{\Gamma,d,\mathbb{s}}[\text{recompute } t]$; this was implicit in Eq. (7.5) but still present as the primal bindings were part of $\mathcal{D}_\Gamma[t]$.

While this works, this implementation of checkpointing is nevertheless "too simple", as it has a few downsides.

1. As primal code is now included in the dual, free variables of that primal code must be preserved. When done ad-hoc as above by including them as explicit bindings in $\mathcal{D}^0_\Gamma[\text{recompute } t]$, multiple 'recompute' terms that reference the same free variable will lead to that variable being stored multiple times (related to a broader problem in CHAD — see also 'Primal storage without duplication' in Section 7.6). An alternative is to add yet another output to the code transformation to communicate upward which elements of $\mathcal{D}[\Gamma]_1$ are actually required in $\mathcal{D}^2_{\Gamma,d,\mathbb{S}}$, so that binding sites (**let**, **case**, etc.) can make precisely those available and no others.

2. As primal values are stored by consumers of those values, not their producers (again, see Section 7.6), writing 'recompute t' will prevent storage of intermediate values in t but not t's result itself. For example, writing 'recompute $2 \cdot z$' in Section 7.4.3 does not prevent storage of the '2', but 'recompute $(2 \cdot z)$' does; however, this latter form has the (potentially) undesirable side effect of also recomputing the multiplication in the reverse pass.[31]

3. Using 'recompute' limits checkpointing to lexical scopes; more general systems may be able to checkpoint around more general intervals. However, we conjecture that unless one wants theoretically optimal checkpointing as in [Siskind and Pearlmutter 2018], lexical checkpointing is generally sufficient.

7.6 Future work

As noted in the introduction to this chapter, much work remains to be done before CHAD can truly be called a practically efficient, general reverse AD algorithm, despite what we have already achieved. We give an overview of future research in this section, at various levels of concreteness. We skip topics that are straightforward applications of existing research, such as the necessity of an array fusion optimisation pass in the implementation and better code generation.

Static multi-hot array cotangents. In Section 7.2.1, we already expanded the "favourable language" for array indexing from the very strict grammar "$a\,!\,e$" for an array *variable* 'a' (recall Idealised Accelerate in Section 7.1) to allowing product projections in the left argument of (!). An important reason why it was difficult to go further is that array one-hots do not behave as monoids in the way we expect of cotangents: there is no zero (what is the index?), nor a plus (what if

[31] In this specific example, it is probably best to go even further and wrap the entire 'build' body in 'recompute', but this does not generalise to larger programs.

the indices are different?). However, if we generalise to multi-hots (a tuple of zero or more index–value pairs, i.e. a sparse array with a statically-known number of non-zero values), we can give a *heterogeneous* plus: just concatenate the tuples. In Section 7.5.1.2, we already introduced a heterogeneous plus for sparse structures; this fits perfectly with multi-hot arrays, as they are also a statically sparse form of array cotangent.

It may be possible to introduce multi-hots as a family of sparsity descriptions for arrays (as in Section 7.5.1.2) and thereby lose not only the grammatical restriction in the left argument of (!), but also the prohibition on nested arrays in the favourable language. To do this, *all* uses of a homogeneous plus need to be eliminated, as well as the need for a zero of every cotangent type.[32] As accumulation is also a use of homogeneous plus, this places restrictions on the level of static sparsity admissible in an accumulator.

Primal storage without duplication. In CHAD, primals are stored whenever a term former requires some of its arguments in the reverse pass. Notably, this is different from a node registering its *own* value to be stored for the reverse pass whenever a node is used more than once, or not at all. For example, consider $x, y, z \vdash t : \mathbb{R}$:

$$\begin{aligned} t = \mathbf{let}\ & a = t_1; b = t_2 \\ \mathbf{in}\ & x \cdot b + y \cdot b + z \cdot b + a \end{aligned}$$

We get $\mathcal{D}^{0S}_{x,y,z}[t] = \mathcal{D}^{0S}_{x,y,z}[t_1] +\!\!+ \mathcal{D}^{0S}_{x,y,z,a}[t_2] +\!\!+ \{x, b\} +\!\!+ \{y, b\} +\!\!+ \{z, b\}$, where the x, y, z, b, b, b bindings arise from the three multiplications storing the primals of their arguments. Clearly, storing b three times is rather pointless; on the other hand, it is beneficial that a is never stored, despite being used.[33]

Simply detecting and eliminating duplicate entries in $\mathcal{D}^{0S}_\Gamma$ is an attractive solution but ultimately imperfect, as the optimal equivalence criterion here (to fully eliminate all duplicate stores) is "is the same value at runtime", which is an undecidable program property.

Looking at AD algorithms more generally, it is usual for operations to log their *result* to the tape, rather than their arguments. This naturally avoids storing values multiple times, and having this storage discipline in CHAD would also improve the handling of 'recompute t' from Section 7.5.1.5 by allowing it to also control the result value of t. However, it has the downside of requiring more care to avoid storing primals that are unused in the dual, including the final top-level program result (which is never used in the dual).

[32] There is a *statically sparse* zero at every cotangent type, namely $\langle\rangle$ with sparsity 'A', but not every concrete cotangent representation, sparse or not, will have a well-defined zero any more.

[33] This assumes the optimisation that $\mathcal{D}^{0S}_\Gamma[t_1 + t_2] = \mathcal{D}^{0S}_\Gamma[t_1] +\!\!+ \mathcal{D}^{0S}_\Gamma[t_2]$ from page 330.

Thus, applying this approach to CHAD not only requires making primal storage the responsibility of producers rather than consumers, but also an analysis for eliding redundant stores (e.g. a to-be-recorded analysis [Hascoët et al. 2005]). It is unclear whether this is a net positive over a very clever redundant storage analysis, even if an optimal such analysis is out of reach.

Justify complexity of selective accumulation. The distribution of variables between Γ_A and Γ_M in Section 7.5.1.1 does not impact correctness, but may have significant effects on performance and complexity. The primary complexity-relevant considerations are:

1. Indexing into an array that resides in Γ_M results in a dense one-hot cotangent, whereas the alternative results in a simple accumulation.

2. The tuples created by bundling cotangents for Γ_M generally cancel against tuple projections higher up the tree, but this cancellation fails if there is control flow in the way.

Our implementation makes what seem to be sensible choices here, but we have no proof that they preserve time complexity; the choices are also likely to not be performance-optimal.

Function types. Function types were supported in Chapters 5 and 6, but not included in the complexity proof and dropped entirely in this chapter. We conjecture that they can be supported with the correct complexity following the recipe in Section 6.8 without breaking the optimisations discussed in this chapter (except possibly Section 7.5.1.3; see also the next point), but we have no implementation to prove this.

More robust store pruning. As mentioned in Section 7.5.1.3, control flow dynamism in the source program (resulting in equivalent dynamism in the output program) breaks transparency of data flow to the compiler, and hence breaks general dead code analyses for eliminating primal stores. An important challenge in applying existing work (e.g. [Hascoët et al. 2005]) here is integrating a cleverer analysis with the AD algorithm in such a way that the algorithm is still total by construction. Specifically, an analysis prior to AD that instructs the algorithm to store only certain values may result in error cases (of missing stores) in the AD algorithm that can only be proved to be unreachable using an external proof. While we already accept something similar in the output program for addition of coproduct cotangents, the code transformation itself is still total in this chapter.

With a properly functioning 'recompute' primitive (see 'Primal storage without duplication' above), an approach could be to generate cunning placements of 'recompute' to effect the desired storage discipline without risking non-totality.

More precise array lifetime modelling. Tracing-based reverse AD methods generally have obviously the correct complexity, and an important reason for this is that the lifetime of arrays — the time from allocation to deallocation — is obvious in the source program for differentiation. After all, a trace (which is the thing actually being differentiated) is a straight-line program without structure or control flow, so every array has a clear construction site, as well as a clear last usage site after which the array can be deallocated. When reversing the data flow, not only does sharing become addition and addition become[34] sharing: additionally, allocation becomes deallocation and deallocation becomes allocation. More concretely: in the reverse pass, the deallocation site of the original array becomes the allocation site of the accumulator for its cotangent, usage sites become accumulation sites into that accumulator, and the allocation site becomes the place where the accumulator is frozen into an immutable array to be used as incoming cotangent for the operation that produced the original array.

In CHAD, things are less clear: the lifetime of a particular array value at runtime is not necessarily evident from the source program in the presence of dynamic control flow, and thus one cannot (always) statically identify the allocation, usage and deallocation sites of an array. The most we seem to be able to do is compute an approximation: shorter lifetime intervals within which we can track a particular array, and outside of which we cannot. For example, if an array is let-bound, then we can crudely count that binding as an "allocation" and the end of the binding's scope as its "deallocation", with presumably some usages in between. Where the array goes afterwards, or perhaps where it came from, we do not know.[35] Our accumulators (introduced by means of **scope**) describe these lifetimes that we did infer, and whenever an array turns out to span multiple of such statically inferred lifetimes, multiple accumulators will be allocated for it at runtime, possibly with a new zero allocated each time. When these additional zeros are dense they compromise the optimality of our complexity, as already discussed in Section 7.2.2.

These statically inferred lifetimes do have an advantage: as they correspond to the lexical structure of the program, a compiler can understand the data flow and optimise at will. We could simulate the tracing-based approach by introducing

[34]The partial derivatives of (+) are 1, and multiplication with 1 is a no-op.

[35]Assuming `malloc-free`-style allocation, the manual equivalent of our current source language. It is naturally possible to instead provide only structured, or scoped, allocation with a primitive that allocates an array for the duration of execution of a single term. While doing so makes array lifetime tracking easy, it would also be a significant reduction in expressivity of the source language.

more dynamism: track array identities in the forward pass, and in the reverse pass allocate an accumulator in some global pool whenever the last use of a particular array is encountered, accumulate into it whenever the array with the appropriate identifier is encountered again, and freeze and use its value when the production site of the array is encountered. With such a program structure, however, data flow of the accumulators is completely obscure to a compiler, probably leading to suboptimal code.

On the other hand, whenever statically inferred, lexical lifetimes of arrays are insufficient, such inference also did not work in the source program. That is: if we could somehow manage to use our compiler-friendly fully-static design for program fragments where arrays can be fully tracked and switch to dynamic identifier-based runtime passing in between those fragments, the potential optimisations lost due to the introduced dynamism may correspond precisely to lost optimisations in the source program, because of precisely the same dynamism. Can we make an unholy marriage between tracing-based AD and CHAD, exploiting the advantages of fully-static, lexically guided differentiation on the program fragments where it is effective, and the advantages of complexity-optimal tracing AD on the control-flow-infested glue between those fragments where fully-static AD does not help anyway?

It is unlikely that such a union, if indeed achievable, would result in a simple algorithm in any sense of the word, and it would also mark a bigger departure from the semantics than we have allowed ourselves so far in this and the previous chapter. Then again, it might also result in fast code with uniform support for a very large input language. The question is whether that is worth the costs.

8 Conclusion

In this thesis, we have taken a close look at two algorithms for reverse AD on functional languages: dual-numbers reverse AD and CHAD. In this chapter, we take some time to compare the two and consider their relative strengths and weaknesses. We also list promising directions of future work.

8.1 Quantitative comparison

Let us first look at our results from a high level. At the start of this thesis, on page 4, we gave a rather ambitious list of properties that we sought for in reverse AD algorithms for functional languages: it had to be simple and provably correct, have the optimal time complexity, produce fast gradient code, work on second-order functional array languages, preserve parallelism and preferably even handle higher-order programs. Analysing the algorithms in this thesis in broad strokes on these points, we arrive at Table 8.1.

The naive dual-numbers reverse AD (DNRAD) algorithm of Chapter 3 is fairly straightforward to analyse semantically: while a denotational model of the mutable arrays used in the final version may be cumbersome, it is not difficult to semantically relate the final algorithm to the original, highly naive version, and

	DNRAD	Bulk DNRAD	CHAD	Efficient CHAD	Fast CHAD
	Ch. 3	Ch. 4	Ch. 5	Ch. 6	Ch. 7
Simplicity / provab.	yes	okay	yes	yes	okay
Correct time cplx.	yes	mostly	no	yes	mostly
Fast in practice	no	okay	no	no	hopefully
SOACs	(yes)	somewhat	yes	yes	yes
Preserves parallelism	somewhat	yes	yes	yes	yes
Higher-order func.	yes	no	yes	yes	probably

Table 8.1: An assessment of the properties we desired in the thesis introduction.

this naive algorithm already has various correctness proofs in the literature. The algorithm also readily generalises to various language features without essentially any work on the part of the implementer, including higher-order functions and recursion and "thus" second-order array combinators (SOACs) — although slowly and sequentially. Task parallelism is supported using a graph of tapes. The downside is that the scalar-focusedness of the algorithm makes data parallelism unavailable and has rather severe consequences for practical performance. Still, if a problem does not involve particularly large collections of scalars and a simple, easily extensible algorithm is desired, the dual-numbers approach can work well.[1]

In a quest to improve the practical performance of dual-numbers reverse AD, we looked at dual arrays in Chapter 4. While rigorous benchmarks for this works are still under way, preliminary results show that performance for array programs is much improved from Chapter 3, as expected, even if the seemingly necessary "unfusion" resulting from the bulk-operation transformation (BOT) limits what we can achieve here. Unfortunately, while the individual components of the algorithm remain relatively simple by themselves, the number of components increases; furthermore, because the BOT is a global program transformation that makes essential use of certain restrictions in its input language, easy extensibility to language features such as higher-order functions is lost. The optimal time complexity is also not attained, strictly speaking, because some one-hot vectors resulting from array indexing may still remain. However, this algorithm does have one (highly pragmatic) advantage: it is available for practical use, right now, in the form of the `horde-ad` Haskell library.

In the second half of the thesis we studied CHAD (Chapter 5) as a reverse AD algorithm with a strong link to theory but a more structure-preserving approach than the dual-numbers algorithm of Chapter 3. This means that arrays are first-class, including higher-order operations on them as well as task and data parallelism. Its time complexity we investigated and fixed in Chapter 6, resulting in the algorithm sporting the highest number of yeses in the table. The one missing 'yes', however — practical performance — is a rather important one.

In the final research chapter of this thesis we set out to fix this, and the algorithm in our implementation (see Section 7.5) can be used to generate fairly good code: if one avoids nested arrays and applies a generous yet thoughtful sprinkling of 'recompute' to work around the duplicate stores, one can derive fairly tight gradient programs from naturally written source programs. Of course, various infelicities clearly remain and the lack of a mature compilation pipeline after AD means that performance of our implementation is still sub-par, but we are hopeful that additional work in this area can result in a genuinely useful implementation of reverse AD.

[1] An example of such an extensible implementation is [Kmett and contributors 2021].

The price of these improvements to CHAD is that the theoretical story becomes more nuanced, as reflected in the reduced number of yeses in the last column of Table 8.1. Although the overall structure still very closely follows naive CHAD from Chapter 5, the algorithm undeniably becomes more complex than it was in Chapters 5 and 6. Furthermore, the structure-rich form of the output code required compromises on the time complexity (Section 7.2.2), and support for function types fell by the wayside. Fortunately, this last point is likely just a result of a lack of time: with a workable representation of the necessary existential types and some solution for (or tolerance of) loss of analysis accuracy around function abstractions, the rules of Chapter 6 may well work as-is.

8.2 Qualitative comparison

Mathematically, both plain dual-numbers reverse AD (of Chapter 3) and CHAD (Chapter 5) arise from a structure-preserving functor from the freely generated category corresponding to their source language to a specific target category supporting the same structure (products, exponentials, etc.). As a result, both are in some sense "unique": the universal property of the source language says that there is but one such functor. The reason why the two algorithms are nevertheless fundamentally different is that they choose different target categories to map into. In particular, dual-numbers (reverse) AD maps into an ordinary simply-typed lambda calculus, changing only the type of scalars and the primitive operations, whereas CHAD maps into a kind of sigma-type category of an ordinary lambda calculus (of primals) and a linear lambda calculus (of (co)tangents) indexed by their primal. (For the latter, see Section 5.2 and [Vákár and Smeding 2022, §5.2, §6].) This fundamentally changes the type transformation and hence the code transformation.

Thus, to understand the qualitative differences between the two algorithms, let us look at the type transformations again. This was previously discussed in Section 3.2 to explain how both conceivably compute the same thing; here, we focus on the effects of the typing on algorithm design and efficiency.

The dual-numbers approach of Chapter 3 has, in basis, the following typing:[2]

$$\Gamma \vdash t : \tau \quad \rightsquigarrow \quad \mathrm{Dual}_c[\Gamma] \vdash \mathcal{R}_{\mathrm{dual}}[t] : \mathrm{Dual}_c[\tau]$$
$$\mathrm{Dual}_c[\mathbb{R}] = \mathbb{R} \times (\underline{\mathbb{R}} \multimap \mathrm{Staged}\, c) \qquad \mathrm{Dual}_c[\mathbb{Z}] = \mathbb{Z}$$
$$\mathrm{Dual}_c[\sigma \times \tau] = \mathrm{Dual}_c[\sigma] \times \mathrm{Dual}_c[\tau]$$

The type transformation only acts non-trivially at the scalar "leaves" of data structures. Correspondingly, language constructs that concern themselves with

[2]The $\mathrm{Dual}_c[\mathbb{R}]$ definition becomes more involved over the course of Chapter 3, but the essence stays the same.

intermediate data structures only (such as $\langle -, - \rangle$, fst, **case**) can be differentiated to themselves: the differentiated data structures carry different content (namely, dual numbers), but they have the same internal structure. This is clearly visible in the first, naive dual-numbers transformation in Fig. 3.6 on page 68, but remains unchanged (apart from monadic lifting) in the later iterations of the algorithm.

As a result, adding support for new language constructs to dual-numbers reverse AD is generally trivial — as long as one does not challenge one of the basic assumptions of the algorithm: that all primitive operations are first-order. In Chapter 4, we not only try to add second-order array operations as primitives, but also require that they have a particular, efficient derivative; this turns out to be too much, resulting in a departure from the compositional nature of the original code transformation by having an additional step before differentiation (the BOT) and after (Delta extraction).

The typing for CHAD is different from dual-numbers AD in an important way, following the additional structure of its target category:[3]

$$\Gamma \vdash t : \tau \quad \rightsquigarrow \quad \mathcal{D}[\Gamma]_1 \vdash \mathcal{R}_{\mathrm{CHAD}}[t] : \mathcal{D}[\tau]_1 \times (\mathcal{D}[\tau]_2 \multimap \mathbf{EVM}\ \mathcal{D}[\Gamma]_2\ \mathbf{1})$$
$$\mathcal{D}[\mathbb{R}]_1 = \mathbb{R} \qquad \mathcal{D}[\mathbb{Z}]_1 = \mathbb{Z} \qquad \mathcal{D}[\sigma \times \tau]_1 = \mathcal{D}[\sigma]_1 \times \mathcal{D}[\tau]_1$$
$$\mathcal{D}[\mathbb{R}]_2 = \mathbb{R} \qquad \mathcal{D}[\mathbb{Z}]_2 = \mathbf{1} \qquad \mathcal{D}[\sigma \times \tau]_2 = \mathbf{1} \sqcup (\mathcal{D}[\sigma]_2 \times \mathcal{D}[\tau]_2)$$

In contrast to $\mathcal{R}_{\mathrm{dual}}[t]$, the return type of $\mathcal{R}_{\mathrm{CHAD}}[t]$ already differs from the type of t at the root of the structure. This means that every language construct, including those that only work on internal structure (such as $\langle -, - \rangle$, etc.), must have a non-trivial derivative that translates the original operation into one on primal–dual pairs; the fact that the primal type is very similar to the original (yet unequal in the case of function types; recall Section 6.8 on page 267) does not diminish this effect. Indeed, this is clearly seen in any of the CHAD code transformations, such as Fig. 6.6 on page 246, where none of the rules are a direct monadic lifting of the original term.

The result is that adding support for a new language construct to CHAD always requires enumerating the (possibly scalar-containing) inputs and outputs of the construct, computing the proper functional reverse derivative for it (in the sense of Section 2.2.2), and then encoding that reverse derivative in terms of the primal–dual pairs.[4] And indeed, where support for higher-order functions was immediate in Chapter 3, their support in CHAD, especially complexity-efficiently, required more thinking (Sections 5.1.4 and 6.8).

[3] We include the modifications of Chapter 6 for completeness and to ease comparison with the (simply-typed) dual-numbers transformation, but they change little about the conclusions here, nor would the additional complications from Chapter 7.

[4] In some ways this is analogous to how we proceeded for differentiating a SOAC in Section 2.2.7.

In return for this required effort, if one happens to already have an efficient implementation of a language construct's reverse derivative, CHAD allows you to set this as the derivative of that construct directly without needing to care about the rest of the algorithm. This is illustrated by the addition of some SOACs to the algorithm in Fig. 6.8 (page 262). We can contrast this with plain dual-numbers reverse AD of Chapter 3, where one needs to interface with the scalar backpropagators in order to add e.g. a SOAC to the language (losing most the efficient bulk properties of the SOAC in the process), and with the dual-arrays approach of Chapter 4, where the necessity of supporting vectorisation with the BOT restricts the kinds of primitive operations that can be added to the language.

Ultimately, functional array languages get their efficiency from the fact that a lot of program structure is evident in the source code: collective operations (with some kind of uniformity in the way they operate on each of the individual array elements) are visible as such to the compiler, and no program analysis needs to be applied to recover that structure. Thus, when differentiating programs in such languages with the desire that the result is still fast, we must ensure that our generated gradient code encodes its own structure in the output code in an effective way. At the very least, we must take care not to break up the structure already present in the input program.

The naive dual-numbers algorithm in Chapter 3 completely destroys this uniformity structure and consequently is slow for array programs. The bulk-operation-aware dual arrays algorithm in Chapter 4 retains the uniformity of individual operations on scalars, but forgets that they belonged together in a single build. CHAD is able to retain all the structure in the source program, and with the improvements in Chapter 7 including the optimised sparsity of Section 7.5.1.2, its output code is also relatively well-structured. Mutable accumulation remains a sore point, but the reality of inverted data flow in reverse AD may well make this inevitable: the ability to do random reads from the context in the source program is considered table stakes, and these random reads turn into random writes (accumulations) in the reverse pass.

8.3 Future work

Automatic differentiation is in somewhat of an odd position: from some perspectives it feels like a solved problem, while from others it does not at all. If one tells a programmer or researcher in machine learning that one works on AD, they will most likely ask why more fundamental research is needed on something that they use daily and works well. This does not mean that no improvement is possible, however: most fast and practically useful reverse AD implementations use tracing or at least a linear tape, with the ones doing structure-preserving AD

either supporting relatively restricted source languages[5] or compromising on time complexity (e.g. [Schenck et al. 2022]). We think there are still good things to find in the space of structure-preserving reverse AD, for example in the style of CHAD but also otherwise: we might expand the space of supported source languages, get closer to the optimal time complexity (without losing efficiency), or further exploit the availability of a structure-rich derivative program with aggressive compiler optimisations. Succeeding with these goals (hopefully resulting in an advance of the state-of-the-art in performance) will likely force a better understanding of AD as a whole, with which we might hope to find simpler algorithms with feasible correctness proofs, thus even further strengthening our grasp of the topic.

In this thesis, we have attempted to effect a little of this increase in understanding, but as usual, this work is far from done; below, we discuss some interesting avenues for future work.

Fast gradients. While the implementation for Chapter 4 (the `horde-ad` library) achieves good performance, gradient code is still interpreted; this results in suboptimal performance in practice, especially for gather and scatter. Independently, the implementation contains a large number of ad-hoc simplification rules to optimise the various inefficient patterns that come out of the BOT as well as AD. For both of these issues, we expect that using an established parallel array language that can express our core language, such as Futhark, as a compilation target can give a significant performance boost to `horde-ad`.

While a good array backend is naturally also a requirement for CHAD, there are also many concrete, as well as less concrete, potential improvements to the algorithm to increase the performance of its output code. These are already discussed in detail in Section 7.6.

CHAD complexity proof. The complexity proof for Efficient CHAD (Chapter 6) covers only the first-order algorithm and restricts itself to the sequential setting. An extension to array combinators and lambda abstraction is warranted, as is some way of describing the parallel span in case those array combinators are run on parallel hardware. Kaler et al. [2021] propose a target span for parallel AD, bounding the additional logarithmic factor by $O(\log(work))$, but we suspect that one can be more precise: the logarithm should only need to capture the array sizes actually in use by the program. However, proving this (and perhaps formulating desired span complexity in a way that does not rely on uncomputable properties of the program, such as internal array sizes) is future work.

[5] For example, while JAX can (currently, version 0.9.0.1) differentiate a general fold, it can do so only for closed functions and on statically known shapes via expansion into a written-out tree reduction.

For Fast CHAD (Chapter 7), complexity-optimality is not provable because it is, unfortunately, false (see Section 7.2.2). Nevertheless, it would be valuable to make more precise what guarantees we *can* provide, as we expect the concessions that we made on this front to be relatively unimportant on practical source programs.

Inductive data types. While some of the algorithms in this thesis can handle both products and coproducts, none are generalised to inductive (let alone recursive) types. Such data structures are uncommon in numerical code, but omitting them is unsatisfying: with products and coproducts supported, we seem so close. There is theory for CHAD in this area [Nunes and Vákár 2023], but no (complexity-)efficient rendition has been constructed yet.

Imperative languages. This thesis focuses on functional languages, but traditionally, performance-sensitive numerical code is written in imperative languages. If the imperativeness of the ambient language is incidental and the computation is actually functional in nature, then this is mostly a question of translation: basic imperative code can be translated via static single-assignment (SSA) form to functional code. [Appel 1998] However, if the program non-trivially uses mutation, especially inside heap allocations, then our methods do not directly apply (without some translation to state-passing style that risks destroying performance utterly).

As noted previously under 'More precise array lifetime modelling' in Section 7.6, (lexically) structured array allocation might still support a structured derivative like that produced by CHAD, but with traditional `malloc-free` memory management or a garbage collector, the AD algorithm will necessarily need to have a dynamic (define-by-run) flavour, at least partly.

In general, one can consider adding various effects — algebraic or not — to the source language; if an effect does not interact with real scalars and it still evident what it means to differentiate a program with such an effect, plain dual-numbers reverse AD (Chapter 3) is likely to apply without much effort. Even for those effects, however, the more efficient algorithms are likely to need significant extensions.

Probabilistic programs. A special kind of effect that interacts non-trivially with differentiation is sampling from a probability distribution. This effect can be used to formalise probabilistic programming languages, which are an important use case for AD. Prior research on how to differentiate it in a modular and principled fashion exists for forward AD (e.g. [Lew et al. 2023]), but it is unclear

how to apply these ideas to reverse AD and in particular structure-preserving approaches such as CHAD.

Index

Bibliography

Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek Gordon Murray, Benoit Steiner, Paul A. Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In Kimberly Keeton and Timothy Roscoe, editors, *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, pages 265–283. USENIX Association, 2016. URL https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi. 1, 10, 41, 115, 270

Mario Alvarez-Picallo, Dan R. Ghica, David Sprunger, and Fabio Zanasi. Functorial string diagrams for reverse-mode automatic differentiation. *CoRR*, abs/2107.13433, 2021. URL https://arxiv.org/abs/2107.13433. 269, 270

Andrew W. Appel. SSA is functional programming. *ACM SIGPLAN Notices*, 33(4): 17–20, 1998. doi: 10.1145/278283.278285. URL https://doi.org/10.1145/278283.278285. 353

David van Balen, Gabriele Keller, Ivo Gabe de Wolff, and Trevor L. McDonell. Fusing gathers with integer linear programming. In *Proceedings of the 1st ACM SIGPLAN International Workshop on Functional Programming for Productivity and Performance*, FProPer 2024, page 10–23, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400711008. doi: 10.1145/3677997.3678227. URL https://doi.org/10.1145/3677997.3678227. 16, 185

Gilles Barthe, Raphaëlle Crubillé, Ugo Dal Lago, and Francesco Gavazzo. On the versatility of open logical relations - continuity, automatic differentiation, and a containment theorem. volume 12075 of *Lecture Notes in Computer Science*, pages 56–83. Springer, 2020. doi: 10.1007/978-3-030-44914-8_3. URL https://doi.org/10.1007/978-3-030-44914-8_3. 270

Walter Baur and Volker Strassen. The complexity of partial derivatives. *Theor.*

Comput. Sci., 22:317–330, 1983. doi: 10.1016/0304-3975(83)90110-X. URL https://doi.org/10.1016/0304-3975(83)90110-X. 33

Atılım Güneş Baydin, Barak A. Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic differentiation in machine learning: a survey. *J. Mach. Learn. Res.*, 18:153:1–153:43, 2017. URL http://jmlr.org/papers/v18/17-468.html. 38, 53

Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R. Newton, Simon Peyton Jones, and Arnaud Spiwack. Linear haskell: practical linearity in a higher-order polymorphic language. *Proc. ACM Program. Lang.*, 2(POPL):5:1–5:29, 2018. doi: 10.1145/3158093. URL https://doi.org/10.1145/3158093. 252

Christian Bischof. Issues in parallel automatic differentiation. 1991. Argonne Preprint MCS-P235-0491. 115

Christian Bischof, Andreas Griewank, and David Juedes. Exploiting parallelism in automatic differentiation. In *Proceedings of the 5th International Conference on Supercomputing*, pages 146–153, 1991. 115

Guy E Blelloch. Nesl: a nested data parallel language. Technical report, Carnegie Mellon University, 1 1992. 186

Guy E. Blelloch. Programming parallel algorithms. *Commun. ACM*, 39(3):85–97, 1996. doi: 10.1145/227234.227246. URL https://doi.org/10.1145/227234.227246. 16

Guillaume Boisseau and Jeremy Gibbons. What you needa know about Yoneda: Profunctor optics and the Yoneda lemma (functional pearl). *Proc. ACM Program. Lang.*, 2(ICFP):84:1–84:27, 2018. doi: 10.1145/3236779. URL https://doi.org/10.1145/3236779. 56, 61

James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: Composable transformations of Python+NumPy programs, 2018. URL http://github.com/jax-ml/jax. 2, 39, 115, 186, 270

Mark R. Brown and Robert Endre Tarjan. A fast merging algorithm. *J. ACM*, 26(2):211–226, 1979. doi: 10.1145/322123.322127. URL https://doi.org/10.1145/322123.322127. 243, 272

Aloïs Brunel, Damiano Mazza, and Michele Pagani. Backpropagation in the simply typed lambda-calculus with linear negation. *Proc. ACM Program. Lang.*,

4(POPL):64:1–64:27, 2020. doi: 10.1145/3371132. URL https://doi.org/10.1145/3371132. 49, 50, 53, 71, 72, 112, 113

Tom H. Brus, Marko C. J. D. van Eekelen, M. O. van Leer, and Marinus J. Plasmeijer. CLEAN: A language for functional graph writing. In Gilles Kahn, editor, *Functional Programming Languages and Computer Architecture, Portland, Oregon, USA, September 14-16, 1987, Proceedings*, volume 274 of *Lecture Notes in Computer Science*, pages 364–384. Springer, 1987. doi: 10.1007/3-540-18317-5_20. URL https://doi.org/10.1007/3-540-18317-5_20. 35

HM Bucker, Bruno Lang, Arno Rasch, Christian H Bischof, and Dieter an Mey. Explicit loop scheduling in openmp for parallel automatic differentiation. In *Proceedings 16th Annual International Symposium on High Performance Computing Systems and Applications*, pages 121–126. IEEE, 2002. 115

Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *J. Funct. Program.*, 19(5):509–543, 2009. doi: 10.1017/S0956796809007205. URL https://doi.org/10.1017/S0956796809007205. 194

Bob Carpenter, Matthew D. Hoffman, Marcus A. Brubaker, Daniel D. Lee, Peter Li, and Michael Betancourt. The Stan Math library: Reverse-mode automatic differentiation in C++. *CoRR*, abs/1509.07164, 2015. URL http://arxiv.org/abs/1509.07164. 38

Bob Carpenter, Andrew Gelman, Matthew D. Hoffman, Daniel Lee, Ben Goodrich, Michael Betancourt, Marcus Brubaker, Jiqiang Guo, Peter Li, and Allen Riddell. Stan: A probabilistic programming language. *Journal of Statistical Software*, 76 (1):1–32, 2017. doi: 10.18637/jss.v076.i01. URL https://www.jstatsoft.org/index.php/jss/article/view/v076i01. 2

Manuel M. T. Chakravarty, Roman Leshchinskiy, Simon L. Peyton Jones, Gabriele Keller, and Simon Marlow. Data parallel Haskell: a status report. In Neal Glew and Guy E. Blelloch, editors, *Proceedings of the POPL 2007 Workshop on Declarative Aspects of Multicore Programming, DAMP 2007, Nice, France, January 16, 2007*, pages 10–18. ACM, 2007. doi: 10.1145/1248648.1248652. URL https://doi.org/10.1145/1248648.1248652. 186

Manuel M. T. Chakravarty, Gabriele Keller, Sean Lee, Trevor L. McDonell, and Vinod Grover. Accelerating Haskell array codes with multicore GPUs. In Manuel Carro and John H. Reppy, editors, *Proceedings of the POPL 2011 Workshop on Declarative Aspects of Multicore Programming, DAMP 2011, Austin, TX, USA, January 23, 2011*, pages 3–14, New York, NY, USA, 2011. ACM. doi: 10.1145/1926354.1926358. 1, 12, 304, 306, 334

Curtis Chin Jen Sem. Formalized correctness proofs of automatic differentiation in Coq. *Master's Thesis, Utrecht University*, 12 2020. URL https://studenttheses.uu.nl/handle/20.500.12932/38375. 49, 270

Adam Chlipala. Parametric higher-order abstract syntax for mechanized semantics. In James Hook and Peter Thiemann, editors, *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008, Victoria, BC, Canada, September 20-28, 2008*, pages 143–156. ACM, 2008. doi: 10.1145/1411204.1411226. URL https://doi.org/10.1145/1411204.1411226. 194

Benjamin Dauvergne and Laurent Hascoët. The data-flow equations of checkpointing in reverse automatic differentiation. In Vassil N. Alexandrov, G. Dick van Albada, Peter M. A. Sloot, and Jack J. Dongarra, editors, *Computational Science - ICCS 2006, 6th International Conference, Reading, UK, May 28-31, 2006, Proceedings, Part IV*, volume 3994 of *Lecture Notes in Computer Science*, pages 566–573. Springer, 2006. doi: 10.1007/11758549_78. URL https://doi.org/10.1007/11758549_78. 40

Paulo Emílio de Vilhena and François Pottier. Verifying an effect-handler-based define-by-run reverse-mode AD library. *Logical Methods in Computer Science*, 19, 2023. 115, 270

Edsko de Vries, Rinus Plasmeijer, and David M. Abrahamson. Uniqueness typing simplified. In Olaf Chitil, Zoltán Horváth, and Viktória Zsók, editors, *Implementation and Application of Functional Languages, 19th International Workshop, IFL 2007, Freiburg, Germany, September 27-29, 2007. Revised Selected Papers*, volume 5083 of *Lecture Notes in Computer Science*, pages 201–218. Springer, 2007. doi: 10.1007/978-3-540-85373-2_12. URL https://doi.org/10.1007/978-3-540-85373-2_12. 35

Conal Elliott. The simple essence of automatic differentiation. *Proc. ACM Program. Lang.*, 2(ICFP):70:1–70:29, 2018. doi: 10.1145/3236765. 61, 205, 269

Conal M. Elliott. Beautiful differentiation. In Graham Hutton and Andrew P. Tolmach, editors, *Proceeding of the 14th ACM SIGPLAN international conference on Functional programming, ICFP 2009, Edinburgh, Scotland, UK, August 31 - September 2, 2009*, pages 191–202. ACM, 2009. doi: 10.1145/1596550.1596579. URL https://doi.org/10.1145/1596550.1596579. 118, 184

Martin Elsman, Fritz Henglein, Robin Kaarsgaard, Mikkel Kragh Mathiesen, and Robert Schenck. Combinatory adjoints and differentiation. In Jeremy Gibbons and Max S. New, editors, *Proceedings Ninth Workshop on Mathematically Structured Functional Programming, MSFP@ETAPS 2022, Munich, Germany, 2nd April*

2022, volume 360 of *EPTCS*, pages 1–26, 2022. doi: 10.4204/EPTCS.360.1. URL https://doi.org/10.4204/EPTCS.360.1. 143, 270

Andy Gill. Type-safe observable sharing in haskell. In Stephanie Weirich, editor, *Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell, Haskell 2009, Edinburgh, Scotland, UK, 3 September 2009*, pages 117–128. ACM, 2009. doi: 10.1145/1596638.1596653. URL https://doi.org/10.1145/1596638.1596653. 190

Andreas Griewank and Andrea Walther. *Evaluating derivatives - principles and techniques of algorithmic differentiation, Second Edition*. SIAM, 2008. ISBN 978-0-89871-659-7. doi: 10.1137/1.9780898717761. URL https://doi.org/10.1137/1.9780898717761. 4, 33, 38, 53, 71, 88, 185

José Grimm, Loïc Pottier, and Nicole Rostaing-Schmidt. Optimal Time and Minimum Space-Time Product for Reversing a Certain Class of Programs. Technical Report RR-2794, INRIA, February 1996. URL https://inria.hal.science/inria-00073896. 43

Laurent Hascoët and Valérie Pascual. The Tapenade automatic differentiation tool: Principles, model, and specification. *ACM Trans. Math. Softw.*, 39(3):20:1–20:43, 2013. doi: 10.1145/2450153.2450158. 42, 115

Laurent Hascoët, Uwe Naumann, and Valérie Pascual. "To be recorded" analysis in reverse-mode automatic differentiation. *Future Gener. Comput. Syst.*, 21(8): 1401–1417, 2005. doi: 10.1016/J.FUTURE.2004.11.009. URL https://doi.org/10.1016/j.future.2004.11.009. 42, 339, 343

W. K. Hastings. Monte Carlo sampling methods using Markov chains and their applications. *Biometrika*, 57(1):97–109, 04 1970. ISSN 0006-3444. doi: 10.1093/biomet/57.1.97. URL https://doi.org/10.1093/biomet/57.1.97. 3

Troels Henriksen. *Design and Implementation of the Futhark Programming Language*. PhD thesis, University of Copenhagen, Universitetsparken 5, 2100 København, 11 2017. 1, 12

Troels Henriksen, Niels G. W. Serup, Martin Elsman, Fritz Henglein, and Cosmin E. Oancea. Futhark: purely functional gpu-programming with nested parallelism and in-place array updates. In Albert Cohen and Martin T. Vechev, editors, *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, pages 556–571. ACM, 2017. doi: 10.1145/3062341.3062354. URL https://doi.org/10.1145/3062341.3062354. 16, 186

Adrian Hill, Guillaume Dalle, and Alexis Montoison. An illustrated guide to automatic sparse differentiation. In *The Fourth Blogpost Track at ICLR 2025*, 2025. URL https://iclr-blogposts.github.io/2025/blog/sparse-autodiff/. 47

Robin J. Hogan. Fast reverse-mode automatic differentiation using expression templates in C++. *ACM Trans. Math. Softw.*, 40(4):26:1–26:16, 2014. doi: 10.1145/2560359. URL https://doi.org/10.1145/2560359. 40

Shuhong Huang, Shizhi Tang, Yuan Wen, Huanqi Cao, Ruibai Tang, Yidong Chen, Jiping Yu, Yang Li, Chao Jiang, Limin Xiao, and Jidong Zhai. Pardiff: Efficiently parallelizing reverse-mode automatic differentiation with direct indexing. In Tony Hosking, Madan Musuvathi, and Kenjiro Taura, editors, *Proceedings of the 31st ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming, PPoPP 2026, Sydney, NSW, Australia, 31 January 2026 - 4 February 2026*, pages 452–465. ACM, 2026. doi: 10.1145/3774934.3786418. URL https://doi.org/10.1145/3774934.3786418. 43

Jan Hückelheim and Laurent Hascoët. Source-to-source automatic differentiation of openmp parallel loops. *ACM Transactions on Mathematical Software (TOMS)*, 48(1):1–32, 2022. 115

Jan Hückelheim, Harshitha Menon, William S. Moses, Bruce Christianson, Paul D. Hovland, and Laurent Hascoët. Understanding automatic differentiation pitfalls. *CoRR*, abs/2305.07546, 2023. doi: 10.48550/ARXIV.2305.07546. URL https://doi.org/10.48550/arXiv.2305.07546. 19, 100

R. John M. Hughes. A novel representation of lists and its application to the function "reverse". *Inf. Process. Lett.*, 22(3):141–144, 1986. doi: 10.1016/0020-0190(86)90059-1. URL https://doi.org/10.1016/0020-0190(86)90059-1. 55, 56, 83, 263

Mathieu Huot, Sam Staton, and Matthijs Vákár. Correctness of automatic differentiation via diffeologies and categorical gluing. In Jean Goubault-Larrecq and Barbara König, editors, *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*, volume 12077 of *Lecture Notes in Computer Science*, pages 319–338. Springer, 2020. doi: 10.1007/978-3-030-45231-5_17. URL https://doi.org/10.1007/978-3-030-45231-5_17. 49, 50, 52, 71, 112, 113, 143, 270

Mathieu Huot, Sam Staton, and Matthijs Vákár. Higher order automatic differentiation of higher order functions. *Log. Methods Comput. Sci.*, 18(1), 2022. doi:

10.46298/LMCS-18(1:41)2022. URL https://doi.org/10.46298/lmcs-18(1:41)2022. 45, 123, 270

Kenneth E. Iverson. *A Programming Language*. Wiley, 1962. 1

Koen Jacobs, Dominique Devriese, and Amin Timany. Purity of an ST monad: Full abstraction by semantically typed back-translation. *Proc. ACM Program. Lang.*, 6(OOPSLA1):1–27, 2022. doi: 10.1145/3527326. URL https://doi.org/10.1145/3527326. 90

Wenzel Jakob, Sébastien Speierer, Nicolas Roussel, and Delio Vicini. Dr.Jit: A just-in-time compiler for differentiable rendering. *Transactions on Graphics (Proceedings of SIGGRAPH)*, 41(4), July 2022. doi: 10.1145/3528223.3530099. 39

Johann Joss. *Algorithmisches Differenzieren.* Doctoral thesis, ETH Zurich, Zürich, 1976. Diss. Math. ETH Zürich, Nr. 5757, 0000. Ref.: Huber, P.J. ; Korref.: Henrici, P. 38

Tim Kaler, Tao B Schardl, Brian Xie, Charles E Leiserson, Jie Chen, Aldo Pareja, and Georgios Kollias. PARAD: A work-efficient parallel algorithm for reverse-mode automatic differentiation. In *Symposium on Algorithmic Principles of Computer Systems (APOCS)*, pages 144–158. SIAM, 2021. 115, 352

Marie Morgane Kerjean and Pierre-Marie Pédrot. ∂ is for Dialectica. In Pawel Sobocinski, Ugo Dal Lago, and Javier Esparza, editors, *Proceedings of the 39th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2024, Tallinn, Estonia, July 8-11, 2024*, pages 48:1–48:13. ACM, 2024. doi: 10.1145/3661814.3662106. URL https://doi.org/10.1145/3661814.3662106. 269

Edward Kmett and contributors. ad: Automatic differentiation, 2021. URL https://hackage.haskell.org/package/ad. (Haskell library on Hackage). 50, 51, 57, 108, 348

Faustyna Krawiec, Simon Peyton Jones, Neel Krishnaswami, Tom Ellis, Richard A. Eisenberg, and Andrew W. Fitzgibbon. Provably correct, asymptotically efficient, higher-order reverse-mode automatic differentiation. *Proc. ACM Program. Lang.*, 6(POPL):1–30, 2022. doi: 10.1145/3498710. URL https://doi.org/10.1145/3498710. 50, 51, 108, 112, 113, 114, 119, 121, 124, 130, 132, 141, 165, 184, 186

John Launchbury and Simon L. Peyton Jones. Lazy functional state threads. In Vivek Sarkar, Barbara G. Ryder, and Mary Lou Soffa, editors, *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation (PLDI), Orlando, Florida, USA, June 20-24, 1994*, pages 24–35.

ACM, 1994. doi: 10.1145/178243.178246. URL https://doi.org/10.1145/178243.178246. 90, 252

John M Lee. *Smooth manifolds*. Springer, 2003. ISBN 978-1-4419-9982-5. doi: 10.1007/978-1-4419-9982-5. 227, 228

Alexander K. Lew, Mathieu Huot, Sam Staton, and Vikash K. Mansinghka. ADEV: sound automatic differentiation of expected values of probabilistic programs. *Proc. ACM Program. Lang.*, 7(POPL):121–153, 2023. doi: 10.1145/3571198. URL https://doi.org/10.1145/3571198. 353

Seppo Linnainmaa. Taylor expansion of the accumulated rounding error. *BIT Numerical Mathematics*, 16(2):146–160, 1976. 27, 38

Dong C. Liu and Jorge Nocedal. On the limited memory BFGS method for large scale optimization. *Math. Program.*, 45(1-3):503–528, 1989. doi: 10.1007/BF01589116. URL https://doi.org/10.1007/BF01589116. 2

Fernando Lucatelli Nunes and Matthijs Vákár. Automatic differentiation for ML-family languages: Correctness via logical relations. *Mathematical Structures in Computer Science*, 34(8):747–806, 2024. doi: 10.1017/S0960129524000215. 49, 52, 56, 71, 98, 100, 113, 143, 270

Dougal Maclaurin. *Modeling, Inference and Optimization with Composable Differentiable Procedures*. PhD thesis, Harvard University, 4 2016. 38

Charles C. Margossian. A review of automatic differentiation and its efficient implementation. *Wiley Interdiscip. Rev. Data Min. Knowl. Discov.*, 9(4), 2019. doi: 10.1002/widm.1305. 38, 40, 46

Damiano Mazza and Michele Pagani. Automatic differentiation in PCF. *Proc. ACM Program. Lang.*, 5(POPL):1–27, 2021. doi: 10.1145/3434309. URL https://doi.org/10.1145/3434309. 71, 100, 113

Trevor L. McDonell, Manuel M. T. Chakravarty, Gabriele Keller, and Ben Lippmeier. Optimising purely functional GPU programs. In Greg Morrisett and Tarmo Uustalu, editors, *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*, pages 49–60. ACM, 2013. doi: 10.1145/2500365.2500595. URL https://doi.org/10.1145/2500365.2500595. 16, 190

Benjamin Meijs. Formal correctness proofs for efficient CHAD. *Master's Thesis, Utrecht University*, 06 2025. URL https://studenttheses.uu.nl/handle/20.500.12932/49742. 232, 260

Duane Merrill and Michael Garland. Single-pass parallel prefix scan with decoupled look-back. *NVIDIA, Tech. Rep. NVR-2016-002*, 2016. 12, 265

Nicholas Metropolis, Arianna W. Rosenbluth, Marshall N. Rosenbluth, Augusta H. Teller, and Edward Teller. Equation of state calculations by fast computing machines. *The Journal of Chemical Physics*, 21(6):1087–1092, 06 1953. ISSN 0021-9606. doi: 10.1063/1.1699114. URL https://doi.org/10.1063/1.1699114. 3

Yasuhiko Minamide, J. Gregory Morrisett, and Robert Harper. Typed closure conversion. In Hans-Juergen Boehm and Guy L. Steele Jr., editors, *Conference Record of POPL'96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, St. Petersburg Beach, Florida, USA, January 21-24, 1996*, pages 271–283. ACM Press, 1996. doi: 10.1145/237721.237791. URL https://doi.org/10.1145/237721.237791. 42, 269, 273

William Moses and Valentin Churavy. Instead of rewriting foreign code for machine learning, automatically synthesize fast gradients. In H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 12472–12485. Curran Associates, Inc., 2020. URL https://proceedings.neurips.cc/paper/2020/file/9332c513ef44b682e9347822c2e457ac-Paper.pdf. 43

Uwe Naumann. Optimal Jacobian accumulation is NP-complete. *Math. Program.*, 112(2):427–441, 2008. doi: 10.1007/s10107-006-0042-z. 45

Radford M Neal. MCMC using Hamiltonian dynamics. *Handbook of markov chain monte carlo*, 2(11):2, 2011. 3

Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, 9 2007. 212, 259

Fernando Lucatelli Nunes and Matthijs Vákár. CHAD for expressive total languages. *Math. Struct. Comput. Sci.*, 33(4-5):311–426, 2023. doi: 10.1017/S096012952300018X. URL https://doi.org/10.1017/s096012952300018x. 43, 61, 205, 227, 228, 229, 269, 353

Fernando Lucatelli Nunes, Gordon Plotkin, and Matthijs Vákár. Unraveling the iterative CHAD. *CrRR*, abs/2505.15002, 2025. doi: 10.48550/ARXIV.2505.15002. URL https://doi.org/10.48550/arXiv.2505.15002. 205, 207

OpenXLA contributors. Accelerated Linear Algebra (XLA), 2022. URL https://github.com/openxla/xla. 1

Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in PyTorch. In *NIPS 2017 Autodiff Workshop: The future of gradient-based machine learning software and techniques*, Red Hook, NY, USA, 2017. Curran Associates, Inc. 2, 10, 38, 57, 115, 186, 270

Adam Paszke, Daniel D. Johnson, David Duvenaud, Dimitrios Vytiniotis, Alexey Radul, Matthew J. Johnson, Jonathan Ragan-Kelley, and Dougal Maclaurin. Getting to the point: Index sets and parallelism-preserving autodiff for pointful array programming. *Proc. ACM Program. Lang.*, 5(ICFP):1–29, 2021a. doi: 10.1145/3473593. URL https://doi.org/10.1145/3473593. 115, 116, 134, 185, 186, 269

Adam Paszke, Matthew J Johnson, Roy Frostig, and Dougal Maclaurin. 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*, pages 13–23, 2021b. 116, 263, 270, 307

Barak A. Pearlmutter and Jeffrey Mark Siskind. Reverse-mode AD in a functional framework: Lambda the ultimate backpropagator. *ACM Trans. Program. Lang. Syst.*, 30(2):7:1–7:36, 2008. doi: 10.1145/1330017.1330018. URL https://doi.org/10.1145/1330017.1330018. 43, 113, 269, 270

Oriol Abril Pla, Virgile Andréani, Colin Carroll, Larry Dong, Christopher Fonnesbeck, Maxim Kochurov, Ravin Kumar, Junpeng Lao, Christian C. Luhmann, Osvaldo A. Martin, Michael Osthege, Ricardo Vieira, Thomas V. Wiecki, and Robert Zinkov. PyMC: a modern, and comprehensive probabilistic programming framework in Python. *PeerJ Comput. Sci.*, 9:e1516, 2023. doi: 10.7717/PEERJ-CS.1516. URL https://doi.org/10.7717/peerj-cs.1516. 2

Gordon D. Plotkin and John Power. Notions of computation determine monads. In Mogens Nielsen and Uffe Engberg, editors, *Foundations of Software Science and Computation Structures, 5th International Conference, FOSSACS 2002. Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002 Grenoble, France, April 8-12, 2002, Proceedings*, volume 2303 of *Lecture Notes in Computer Science*, pages 342–356. Springer, 2002. doi: 10.1007/3-540-45931-6_24. URL https://doi.org/10.1007/3-540-45931-6_24. 245

Gordon D. Plotkin and Matija Pretnar. Handling algebraic effects. *Log. Methods Comput. Sci.*, 9(4), 2013. doi: 10.2168/LMCS-9(4:23)2013. URL https://doi.org/10.2168/LMCS-9(4:23)2013. 247

Alexey Radul, Adam Paszke, Roy Frostig, Matthew J. Johnson, and Dougal Maclaurin. You only linearize once: Tangents transpose to gradients. *Proc. ACM Program. Lang.*, 7(POPL):1246–1274, 2023. doi: 10.1145/3571236. URL https://doi.org/10.1145/3571236. 186, 269

Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman P. Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In Hans-Juergen Boehm and Cormac Flanagan, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, pages 519–530. ACM, 2013. doi: 10.1145/2491956.2462176. URL https://doi.org/10.1145/2491956.2462176. 1

John C. Reynolds. The discoveries of continuations. *LISP Symb. Comput.*, 6(3-4): 233–248, 1993. 39

John C. Reynolds. Definitional interpreters for higher-order programming languages. *High. Order Symb. Comput.*, 11(4):363–397, 1998. doi: 10.1023/A:1010027404223. 96, 268

Robert Schenck, Ola Rønning, Troels Henriksen, and Cosmin E. Oancea. AD for an array language with nested parallelism. In Felix Wolf, Sameer Shende, Candace Culhane, Sadaf R. Alam, and Heike Jagode, editors, *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis, Dallas, TX, USA, November 13–18, 2022*, pages 58:1–58:15. IEEE, 2022. doi: 10.1109/SC41404.2022.00063. URL https://doi.org/10.1109/SC41404.2022.00063. 43, 116, 269, 270, 307, 315, 352

Sven-Bodo Scholz. Single assignment C: efficient support for high-level array operations in a functional setting. *J. Funct. Program.*, 13(6):1005–1059, 2003. doi: 10.1017/S0956796802004458. URL https://doi.org/10.1017/S0956796802004458. 1, 12

Amir Shaikhha, Andrew Fitzgibbon, Dimitrios Vytiniotis, and Simon Peyton Jones. Efficient differentiable programming in a functional array-processing language. *Proc. ACM Program. Lang.*, 3(ICFP):97:1–97:30, 2019. doi: 10.1145/3341701. 49, 51, 114, 270

Amir Shaikhha, Mathieu Huot, and Shideh Hashemian. ∇sd: Differentiable programming for sparse tensors. *arXiv preprint arXiv:2303.07030*, 2023. 271

Tim Sheard and Simon L. Peyton Jones. Template meta-programming for Haskell. *ACM SIGPLAN Notices*, 37(12):60–75, 2002. doi: 10.1145/636517.636528. URL https://doi.org/10.1145/636517.636528. 108

Jesse Sigal. *Automatic Differentiation via Effects and Handlers*. PhD thesis, University of Edinburgh, 06 2024. 39, 115

Jeffrey Mark Siskind and Barak A. Pearlmutter. Efficient implementation of a higher-order language with built-in AD. *CoRR*, abs/1611.03416, 2016. 270

Jeffrey Mark Siskind and Barak A. Pearlmutter. Divide-and-conquer checkpointing for arbitrary programs with no user annotation. *Optimization Methods and Software*, 33(4-6):1288–1330, 2018. doi: 10.1080/10556788.2018.1459621. 40, 43, 88, 341

Tom Smeding. Reverse automatic differentiation for Accelerate. *Master's Thesis, Utrecht University*, 01 2021. https://dspace.library.uu.nl/handle/1874/401621. 306

Tom Smeding and Matthijs Vákár. Efficient dual-numbers reverse AD via well-known program transformations. *CoRR*, abs/2207.03418v2, 2022. 89

Tom Smeding and Matthijs Vákár. Efficient dual-numbers reverse AD via well-known program transformations. *Proc. ACM Program. Lang.*, 7(POPL):1573–1600, 2023a. doi: 10.1145/3571247. URL https://doi.org/10.1145/3571247. 5, 49, 50

Tom Smeding and Matthijs Vákár. Artifact for Efficient CHAD, 10 2023b. URL https://zenodo.org/record/10015321. Artifact for Chapter 6. 259

Tom Smeding and Matthijs Vákár. Efficient CHAD. *Proc. ACM Program. Lang.*, 8(POPL):1060–1088, 2024. doi: 10.1145/3632878. URL https://doi.org/10.1145/3632878. 5, 61, 64, 112, 205, 231

Tom Smeding, Mikołaj Konarski, Simon Peyton Jones, and Andrew Fitzgibbon. Dual-numbers reverse AD for functional array languages. 2025. URL https://arxiv.org/abs/2507.12640. 5, 117

Tom J. Smeding and Matthijs I. L. Vákár. Parallel dual-numbers reverse AD. *Journal of Functional Programming*, 35:e16, 2025. doi: 10.1017/S0956796825100051. 5, 49

Bert Speelpenning. Compiling fast partial derivatives of functions given by algorithms. Technical report, Illinois University, 1 1980. 38

Sam Staton. Completeness for algebraic theories of local state. In C.-H. Luke Ong, editor, *Foundations of Software Science and Computational Structures, 13th International Conference, FOSSACS 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*, volume 6014 of *Lecture Notes in Computer Science*, pages 48–63. Springer, 2010. doi: 10.1007/978-3-642-12032-9_5. URL https://doi.org/10.1007/978-3-642-12032-9_5. 245

Michel Steuwer, Toomas Remmelg, and Christophe Dubach. Lift: a functional data-parallel IR for high-performance GPU code generation. In Vijay Janapa Reddi, Aaron Smith, and Lingjia Tang, editors, *Proceedings of the 2017 International Symposium on Code Generation and Optimization, CGO 2017, Austin, TX, USA, February 4-8, 2017*, pages 74–85. ACM, 2017. URL http://dl.acm.org/citation.cfm?id=3049841. 1, 12

Matthijs Vákár. Reverse AD at higher types: Pure, principled and denotationally correct. In Nobuko Yoshida, editor, *Programming Languages and Systems*, volume 12648 of *Lecture Notes in Computer Science*, pages 607–634. Springer, 2021. doi: 10.1007/978-3-030-72019-3_22. URL https://doi.org/10.1007/978-3-030-72019-3_22. 269

Matthijs Vákár and Tom Smeding. CHAD: Combinatory Homomorphic Automatic Differentiation. In *Proc. TOPLAS 2022*, volume 44, pages 20:1–20:49. ACM, 2022. doi: 10.1145/3527634. URL https://doi.org/10.1145/3527634. 5, 43, 61, 205, 216, 228, 235, 249, 269, 349

Birthe Van den Berg, Tom Schrijvers, James McKinna, and Alexander Vandenbroucke. Forward-or reverse-mode automatic differentiation: What's the difference? *Science of Computer Programming*, 231:103010, 2024. 270

Todd Veldhuizen. Expression templates. Technical report, 6 1995. 40

Dimitrios Vytiniotis, Dan Belov, Richard Wei, Gordon Plotkin, and Martin Abadi. The differentiable curry. In *Program Transformations for ML Workshop at NeurIPS 2019*, 2019. URL https://openreview.net/forum?id=ryxuz9SzDB. 43, 205, 269, 270

Matthijs Vákár. Denotational correctness of forward-mode automatic differentiation for iteration and recursion. *CoRR*, abs/2007.05282, 2020. URL https://arxiv.org/abs/2007.05282. 270

Philip Wadler. Linear types can change the world! In Manfred Broy and Cliff B. Jones, editors, *Programming concepts and methods: Proceedings of the IFIP Work-*

ing Group 2.2, 2.3 Working Conference on Programming Concepts and Methods, Sea of Galilee, Israel, 2-5 April, 1990, page 561. North-Holland, 1990. 35

Fei Wang, James M. Decker, Xilun Wu, Grégory M. Essertel, and Tiark Rompf. Backpropagation with callbacks: Foundations for efficient and expressive differentiable programming. In Samy Bengio, Hanna M. Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, 3-8 December 2018, Montréal, Canada*, pages 10201–10212, 2018. 39, 270

Fei Wang, Daniel Zheng, James M. Decker, Xilun Wu, Grégory M. Essertel, and Tiark Rompf. Demystifying differentiable programming: Shift/reset the penultimate backpropagator. *Proc. ACM Program. Lang.*, 3(ICFP):96:1–96:31, 2019. doi: 10.1145/3341700. 115, 270

R. E. Wengert. A simple automatic derivative evaluation program. *Commun. ACM*, 7(8):463–464, 1964. doi: 10.1145/355586.364791. 38

Sam Westrick, Matthew Fluet, Mike Rainey, and Umut A. Acar. Automatic parallelism management. *Proc. ACM Program. Lang.*, 8(POPL):1118–1149, 2024. doi: 10.1145/3632880. URL https://doi.org/10.1145/3632880. 57

Philip Wolfe. Checking the calculation of gradients. *ACM Trans. Math. Softw.*, 8(4):337–343, 1982. doi: 10.1145/356012.356013. URL https://doi.org/10.1145/356012.356013. 33

Ivo Gabe de Wolff, David P. van Balen, and Gabriele K. Keller. Scheduling task and data parallelism in array languages with work assisting. In Wolfgang E. Nagel, Diana Goehringer, and Pedro C. Diniz, editors, *Euro-Par 2025: Parallel Processing - 31st European Conference on Parallel and Distributed Processing, Dresden, Germany, August 25-29, 2025, Proceedings, Part I*, volume 15900 of *Lecture Notes in Computer Science*, pages 38–52. Springer, 2025. doi: 10.1007/978-3-031-99854-6_3. URL https://doi.org/10.1007/978-3-031-99854-6_3. 13

Nederlandse Samenvatting

Nederlandse samenvatting van dit prachtige werk.
(Dutch layman's summary of the work; to be written.)

IPA Dissertation Series

List that is way too long