Utrecht University

Graduate School of Natural Sciences

Computing Science

Reverse Automatic Differentiation for Accelerate

Author
Tom J. Smeding

Supervisors
Dr. Trevor L. McDonell
Dr. Matthijs I.L. Vákár

January 29, 2021

ICA-6588956

Abstract

Purely functional array programming languages are powerful tools for expressing data-parallel computation in a way that is easily optimised to have good performance on parallel hardware like GPUs. Probabilistic inference and optimisation are fields that need this computational power, but also require *reverse-mode automatic differentiation (AD)* of code written in the programming language. Accelerate is a purely functional array programming language with an optimising compiler for multicore CPU and GPU backends that uses a strongly-typed abstract syntax tree to ensure that no compiler pass can perform type-incorrect program transformations. However, due to the *second-order* nature of the Accelerate language, existing algorithms for reverse-mode AD do not directly apply to Accelerate.

We define a new, pragmatic source-code transformation algorithm for reverse-mode AD on a significant subset of Accelerate; the supported subset excludes loops. We present an implementation of our algorithm in the Accelerate compiler and show that it performs reasonably competitively with other implementations of reverse-mode AD on benchmarks; as a compiler pass in the Accelerate compiler, our implementation cannot perform type-incorrect program transformations. Our reverse-mode AD algorithm can possibly be generalised to other second-order, purely functional array programming languages.

Contents

Acknowledgements

It is an honour and a delight to have supervisors as friendly, helpful and critical as dr. Trevor McDonell, dr. Matthijs Vákár and—unofficially—prof. dr. Gabriele Keller. Thank you for many useful discussions and for showing me how fun research can be.

1 Introduction

High school math teaches us how to differentiate mathematical functions, such as the two-argument function $f = \lambda(x, y).\ x^2 + 7xy^3$. This process results in a different function, in this case $f' = \lambda(x, y).\ (2x + 7y^3, 21xy^2)$, that returns the *gradient* (the vector of partial derivatives) of f at a particular input point.

However, high school math does not teach us how to differentiate a *computer program* that computes some mathematical function: computer programs can contain variable bindings, non-linear control flow, function abtractions, and sometimes more. Methods for automatically differentiating numeric functions written in a programming language are aptly called *automatic differentiation*, abbreviated as AD.

The need for AD arises in various contexts: a well-known application is optimisation in machine learning, where the derivative of a neural network is usually computed with a bespoke *backpropagation* algorithm. However, applications exist also in statistics: maximum likelihood estimation for complex probabilistic models (e.g. specified in a probabilistic programming language) can use gradient-based optimisation, for which derivatives of the probability density function are required. Furthermore, algorithms for approximate Bayesian inference often also require gradients of complicated density functions. Examples of such algorithms are ADVI [30], which is a technique to automate derivation of variational inference algorithms that use AD to compute gradients, and state-of-the-art Markov integration methods such as Hamiltonian Monte Carlo (HMC [40]) and its extension, the No-U-Turn Sampler (NUTS [26]), which use gradients of probability density functions for more effective sampling.[1]

Automatic differentiation comes in two predominant forms: forward-mode AD and reverse-mode AD. If the language on which we wish to perform AD supports operator overloading, forward-mode AD can often be implemented using a relatively simple library written in the language itself; however, for computing the gradient of a function, *reverse-mode AD* is usually much more efficient (see Section 2.1). Unfortunately, implementing reverse-mode AD with good performance is more difficult than doing the same for forward-mode AD, since it requires a form of running the to-be-differentiated program in reverse.

There are multiple possible implementation techniques for reverse-mode AD (see Section 2.1.3), from very dynamic ones (e.g. recording executed operations and interpreting the produced list in reverse) to more static ones (performing a source-code transformation[2] that actively rearranges control flow). Dynamic approaches have the advantage of being more general in the kinds of programming languages they support, which can make them more flexible in actual use; examples of such implementations are Autograd [31] and PyTorch [41]. For compiled languages, on the other hand, static approaches that perform a source-code transformation have the advantage of exposing all of the resulting computation and control flow to the compiler. This allows us to benefit from all optimisations that the compiler implements, and it also avoids any interpretation overhead at runtime. Examples of implementations that take the more static approach are Theano [4], Tapenade [24] and the AD functionality in Tensorflow [1]. A downside of static approaches is that reversing control flow statically requires either a custom compiler plugin or preprocessor that performs a source-code transformation, or a language that has very expressive metaprogramming facilities (for example control operators, as used in e.g. [58]).

Applications using automatic differentiation tend to work with functions that take many param-

[1] This is all the more relevant in 2021: the British government has used the probabilistic programming language Stan [7] to formulate and simulate their COVID-19 epidemic models [20]. Stan uses AD to compute the gradient of probability density functions to support its HMC integrator, which uses the NUTS algorithm.

[2] The term "source-code transformation" is used more generically than the words would suggest: an implementation that works on the internal program representation in a compiler is also said to be a source-code transformation, even if it does not actually operate on the textual source.

eters or work on large amounts of data. Typically, those programs are also parallel in nature, meaning that we would like to leverage modern parallel hardware platforms, such as GPUs, to help run those programs efficiently. An alternative to manually parallising those programs is to write them in a parallel array language that is built from expressive, yet well-parallelisable building blocks that allow writing highly parallel programs in a natural way. In particular, purely functional parallel array languages are very attractive for this purpose: being purely functional allows easier program analysis by the compiler,[3] including automatic parallelisation and advanced loop fusion optimisations that improve performance on parallel hardware.[4] Examples of such languages are Accelerate [8] and Futhark [25]. Having these optimisation frameworks at our disposal when writing parallel array-oriented programs is very beneficial, because manually writing efficient code for GPUs is still quite hard.

From all of the above we conclude that what we want is an implementation of reverse-mode AD, preferably as a source-code transformation, for a practical functional array language that compiles to GPU code. While there is ongoing work that aims to make such an implementation,[5] we are not aware of an implementation that is already mature and readily usable. In this thesis report, we propose an algorithm and provide an implementation for performing efficient reverse-mode AD on a significant subset of the Accelerate programming language.

Our algorithm is indeed in the form of a source-code transformation: given (a parsed representation of) the source code of an Accelerate program, it returns a different Accelerate program that computes the gradient function of the input program. The transformation result uses no mutation and is expressed in unmodified Accelerate, a purely functional language. We note that our algorithm can potentially be generalised to other functional array languages that have a similar structure to the Accelerate language. We implement the transformation in the Accelerate compiler, which not only allows us to provide a natural interface to the programmer, but also means that the transformed result program is subject to the same high-level optimisations in the Accelerate compiler as the original program.

Furthermore, the internal program representation in the Accelerate compiler is a *strongly-typed* abstract syntax tree, meaning that no compiler pass can produce type-incorrect code. While this does not guarantee semantic correctness, it rules out a large class of bugs in the implementation of the AD transformation.

In this project, we are guided by the following main research question:

- How can a reverse-mode AD system for Accelerate using source-code transformation provide performance competitive with existing fast AD frameworks?

Important components of this main question are the following subquestions:

1. What additional language primitives, on top of those in the current Accelerate language, are needed to express the gradient computation code for an Accelerate program?
2. How does one perform non-local full-program transformations (in particular, reverse-mode AD) on a strongly-typed AST?
3. What optimisations, both generic compiler optimisations and improvements to the AD algorithm, are necessary for competitive performance on top of a naive implementation of AD?

[3] It also allows easier program analysis by the *programmer*, which is an additional benefit.

[4] The advantages of purely functional languages for parallel computation are also appearing in industry, considering the popularity of frameworks like MapReduce [16] and TensorFlow.

[5] The authors of both the Futhark language and the Dex language [32] are working on a reverse-mode AD implementation in their respective compilers; in both cases, this is ongoing and yet unpublished work. Both developments were (partially) in parallel with this thesis project.

1.1 Contributions

In this report, we present the following contributions:

- A source-code transformation algorithm for reverse-mode automatic differentiation on an Accelerate-like language, that covers most of the features present in Accelerate except for loops.
- A working, usable implementation of this algorithm in the Accelerate compiler.
- Benchmarks showing that the implementation produces reasonably competitive code.
- Experiences and conclusions from building the implementation on a strongly-typed AST.

Unfortunately, due to time constraints, we were unable to do any significant optimisation on the implementation after building it. Therefore, subquestion 3 will receive only a limited answer in this report.

1.2 Report structure

First, in Section 2, we give background on the topics of automatic differentiation and functional array languages, including a survey of relevant literature.

Then, in Section 3, we describe a model language that slightly simplifies the internal representation of the Accelerate language in the compiler, and we define a reverse-mode AD algorithm on that language. We give examples to illustrate the workings of the algorithm.

Afterwards, in Section 4, we describe the important observations made while writing the implementation, focusing on those parts where the implementation requires some extra creativity not specified by the algorithm. In this section we also show how the implementation was tested in order to instill some confidence as to its correctness.

In Section 5, we present benchmark programs to show that the implementation (together with the Accelerate compiler) produces performant code.

Finally, in Section 6, we provide partial answers to the research questions, and provide reflections and future work. Additionally, we will compare our work with existing, related work.

2 Background

The main aim of this thesis project is to implement a reverse-mode automatic differentiation transformation in the Accelerate compiler. This touches two distinct areas of computer science: the theory and practice of automatic differentiation (AD), and the study of functional programming, including compiler theory, type-safe data structures, and functional array languages. These two areas also have mostly disjoint collections of literature, and hence we will cover them separately in the following sections.

2.1 Automatic differentiation

Automatic differentiation aims to efficiently differentiate numerical functions written in a programming language. Such functions are generally defined on floating-point numbers (in order to approximate real numbers), may use dynamic control flow (conditionals, loops), and can have multiple inputs and multiple outputs. It is generally assumed that the function, though it may invoke side effects, is still a pure function of its inputs, in the sense that given the same inputs, it should give the same outputs. Under this assumption, we can think of the program as a mathematical function $f : \mathbb{R}^m \to \mathbb{R}^n$, for some $m, n \in \mathbb{Z}_{\geq 0}$.

The total derivative of such a function at a particular input point $\vec{x} \in \mathbb{R}^m$ is the Jacobian matrix at that point: $Df(\vec{x}) \in \mathbb{R}^{n \times m}$. Sometimes, what we are interested in is indeed the full Jacobian, but automatic differentiation usually focuses on the reduced problem of either computing one column of the Jacobian (*forward-mode AD*, or *forward AD*) or one row of the Jacobian (*reverse-mode AD*, or *reverse AD*). Why we look only at these reduced problems and not at generating the full Jacobian, we will find out after we discuss *how* AD computes these parts of the Jacobian.

A mathematical function defined by a program computes its result in multiple steps. Each step modifies the state, or possibly a small part of it. It is useful to see a step as a *function* that maps the previous state to the next one, starting with the function input as the initial state and ending with the function output as the last state. Consider as an example the function $g : \mathbb{R}^2 \to \mathbb{R}^2$, defined by the following (informal) program that computes the sum and the product of its two inputs:[6]

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

We can see g as being the composition of three smaller functions, each having their own respective Jacobian:

$$g_3 = \lambda(x, y, s, p).\ (s, p) \qquad g_2 = \lambda(x, y, s).\ (x, y, s, x \cdot y) \qquad g_1 = \lambda(x, y).\ (x, y, x + y)$$

$$Dg_3 = \lambda(x, y, s, p).\ \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad Dg_2 = \lambda(x, y, s).\ \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ y & x & 0 \end{pmatrix} \quad Dg_1 = \lambda(x, y).\ \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \end{pmatrix}$$

Indeed, $g_3 \circ g_2 \circ g_1 = g$. The full Jacobian $Dg((x, y))$ is the matrix obtained by multiplying the individual matrices shown above, evaluated at the arguments at which the g_i are evaluated in a normal execution of g. That is to say:

$$Dg((x, y)) = Dg_3(g_2(g_1((x, y)))) \cdot Dg_2(g_1((x, y))) \cdot Dg_1((x, y))$$

In fact, this is a special case of the *chain rule* for derivatives, which in general can be formulated[7] as $D(g \circ f)(x) = Dg(f(x)) \cdot Df(x)$.

[6] The informal language we use in the example here is meant to be a generic functional programming language that can work with floating-point numbers and tuples of them.

[7] High school math may teach the chain rule with notation more similar to $(f(x))' = f'(x) \cdot x'$. Note that we only have to make this "x" a function of another argument to retrieve the rule $((f \circ h)(y))' = f'(h(y)) \cdot h'(y)$, which is suspiciously similar to our notation. Indeed, generalisation from scalar functions to multi-argument multi-result functions gives the rule used here.

However, a matrix $M \in \mathbb{R}^{n \times m}$ can also be represented as a linear function, in two different ways: one is $\lambda v.\ Mv$, which takes a column vector of size m and left-multiplies M, and the other is $\lambda v.\ vM$, which takes a row vector of size n and right-multiplies M. An important observation is that if $v \in \mathbb{R}^m$ contains a single "1" at position i and only zeros elsewhere, then Mv is precisely the i'th column of M. Likewise, for $v \in \mathbb{R}^n$ with a single "1" at position i, $v^\top M$ is precisely the i'th row of M.

Suppose that we now want to do *forward AD* on g, i.e. we want to compute column i of the Jacobian Dg. That column is equal to the following matrix–vector multiplication:

$$Dg((x,y)) \cdot e_i \;=\; Dg_3(g_2(g_1((x,y)))) \cdot Dg_2(g_1((x,y))) \cdot Dg_1((x,y)) \cdot e_i$$

where $e_i \in \mathbb{R}^2$ is the column vector with a "1" on position i and zeros elsewhere. Because matrix multiplication is associative, we may execute these multiplications in any order we want and get the same result. However, in terms of number of arithmetic operations (scalar addition and multiplication) executed, it is usually beneficial[8] to associate this expression to the right, making sure that we only perform matrix–vector multiplications.

Define, for each $g_i : \mathbb{R}^{n_1} \to \mathbb{R}^{n_2}$, an associated derivative function $g_i^{\text{fwd}} : (\mathbb{R}^{n_1}, \mathbb{R}^{n_1}) \to (\mathbb{R}^{n_2}, \mathbb{R}^{n_2})$ that is semantically equivalent to $\lambda(x,d).\ (g_i(x), Dg_i(x) \cdot d)$. We say "semantically equivalent" because its exact implementation may not involve the matrix–vector multiplication. These functions take an input x and the i'th column of the Jacobian of the preceding part of the program, and return the original output of the corresponding g_i on the input x as well as the i'th column of the Jacobian of the program including this step. We then see that these derivative functions compose to give the derivative function of a larger program: $g^{\text{fwd}} = g_3^{\text{fwd}} \circ g_2^{\text{fwd}} \circ g_1^{\text{fwd}}$, where we define g^{fwd} analogously to g_i^{fwd}.

Suppose now that instead, we want to do *reverse AD* on g. The goal is now to compute row i (not column i) of the Jacobian Dg, which is equal to the following vector–matrix multiplication:

$$e_i^\top \cdot Dg((x,y)) \;=\; e_i^\top \cdot Dg_3(g_2(g_1((x,y)))) \cdot Dg_2(g_1((x,y))) \cdot Dg_1((x,y))$$

where $e_i^\top$ is the 2-dimensional row vector with a "1" on position i and zeros elsewhere. Just like before, to compute this product efficiently, it is usually beneficial in terms of number of arithmetic operations executed to associate this expression to the *left*, making sure to only perform vector–matrix multiplications.

In this case, however, it is unclear if there are g_i^{rev} that, like g_i^{fwd} for forward AD, compose to compute the function value simultaneously with a row of the Jacobian. The reason is that for forward AD, we chose the multiplication order to be in line with the natural execution order of the program (g_1, g_2, g_3), whereas here we are traversing the program in reverse! Indeed, while we can write a function in our informal language that computes the full result $g(x)$ and its Jacobian row $v \cdot Dg(x)$ from an input x and a row-vector v, it would look like this:

$$\begin{aligned} g^{\text{rev}} = \lambda(x,v).\ &\textbf{let } x_1 = g_1(x) \\ &\textbf{in let } x_2 = g_2(x) \\ &\textbf{in let } x_3 = g_3(x) \\ &\textbf{in let } d_3 = v \cdot Dg_3(x_2) \\ &\textbf{in let } d_2 = d_3 \cdot Dg_2(x_1) \\ &\textbf{in let } d_1 = d_2 \cdot Dg_1(x) \\ &\textbf{in } (x_3, d_1) \end{aligned} \tag{1}$$

[8] To be more precise, it is always beneficial in case none of the matrices Dg_i are vectors, i.e. all have at least 2 cells along each side. However, this is the usual case in AD. In general, finding an optimal order (association) for the multiplication of a list of matrices is NP-complete [39].

Note that for a function $f : \mathbb{R}^m \to \mathbb{R}^n$, we had $f^{\text{fwd}} : (\mathbb{R}^m, \mathbb{R}^m) \to (\mathbb{R}^n, \mathbb{R}^n)$, while for reverse AD we have $f^{\text{rev}} : (\mathbb{R}^m, (\mathbb{R}^n)^*) \to (\mathbb{R}^n, (\mathbb{R}^m)^*)$. Here we write V^* for the vector space that is the same as V, but with *row vectors* instead of column vectors.[9]

Contrary to the situation with forward AD, g^{rev} is split in a forward pass (called the *primal pass*) and a backward pass (called the *dual pass*) through g. While we could also have written g^{fwd} in this manner, it was not *necessary*; here it is, because the first computations in the dual pass depend on the last results in the primal pass. It turns out that the necessity of this split is the hallmark of reverse AD, and is the reason for many of its characteristics, including the relative implementation difficulty as compared to forward AD.

2.1.1 What does AD compute?

Above, we gave two functions, $g^{\text{fwd}} = g_3^{\text{fwd}} \circ g_2^{\text{fwd}} \circ g_1^{\text{fwd}}$ and g^{rev} in Eq. (1), that respectively (in addition to the normal function output of g) compute a Jacobian–vector product and a vector–Jacobian product with the Jacobian of g. The Jacobian of a function $f : \mathbb{R}^m \to \mathbb{R}^n$ looks as follows:

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

We can see that column i of Df is the column vector $\left(\frac{\partial f_1}{\partial x_i}, \ldots, \frac{\partial f_n}{\partial x_i}\right)^\top$. This contains the derivatives of all output values with respect to the i'th input value, or in other words, the *tangent vector* of f with respect to x_i. In a sense, this is the sensitivity of f to changes in the value of the input x_i. The individual derivatives in this vector are called *tangents*: $\frac{\partial f_j}{\partial x_i}$ is the *tangent* of f_j with respect to x_i. Therefore, passing the column vector e_i, where $e_1 = (1, 0)^\top$ and $e_2 = (0, 1)^\top$, as the second argument to g^{fwd} returns the tangent vector $\left(\frac{\partial g_1}{\partial x_i}, \frac{\partial g_2}{\partial x_i}\right)^\top$.

On the other hand, row i of Df is the row vector $\left(\frac{\partial f_i}{\partial x_1}, \ldots, \frac{\partial f_i}{\partial x_m}\right)$, which is the *gradient* of f_i with respect to the input vector. The individual components of this vector are called *adjoints*: $\frac{\partial f_j}{\partial x_i}$ is, in addition to the tangent of f_j with respect to x_i, also called the *adjoint*[10] of x_i in the context of computing f_j. This terminology of adjoints will come back later when we discuss reverse AD specifically. As an example, passing the row vector $e_i^\top$ to g^{rev} computes the gradient vector $\left(\frac{\partial g_i}{\partial x_1}, \frac{\partial g_i}{\partial x_2}\right)$.

By passing a vector to the function resulting from forward (respectively reverse) AD that does not contain only a single "1" but multiple nonzero entries, the result is the linear combination of columns (respectively rows) from the Jacobian with coefficients from the supplied vector. Indeed, the case with a single "1" is a special case of this linear combination.

Why not the full Jacobian? As announced at the beginning of Section 2.1, in AD we often restrict the problem of computing the full Jacobian to computing only a single column or row; afterwards we described a way in which we can compute one such column or row. The question remains, however, why it is useful to restrict the problem in this way.

Both g^{fwd} and g^{rev} perform all the steps of the original function g (namely, g_1, g_2 and g_3), as well as, for each step, one multiplication of the Jacobian matrix of that step with a vector. These

[9] The V^* notation is really the vector space dual; however, the further theoretical implications of that are out of scope in this work.

[10] The word *adjoint* has many meanings in mathematics. The meaning as used here and in the AD literature, however, is completely unrelated to most other meanings; as far as we know, it is in particular also unrelated to its meaning in category theory.

single-step Jacobian matrices are specific to the arithmetic operations performed in those steps, and can often be written more concisely using the derivative of the arithmetic operation at hand.

For example, consider $g_2^{\text{fwd}}(x, d)$, which was defined using $Dg_2(x)$ but can be rewritten as follows:[11]

$$\begin{aligned} g_2^{\text{fwd}}(x,d) &= \lambda(x,d).\ (g_2(x), Dg_2(x) \cdot d) \\ &= \lambda((x,y,s),(xt,yt,st)).\ \left((x,y,s,x\cdot y), \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ y & x & 0 \end{pmatrix} \cdot d\right) \\ &= \lambda((x,y,s),(xt,yt,st)).\ ((x,y,s,x\cdot y),(xt,yt,st,y\cdot xt + x\cdot yt)) \end{aligned}$$

If we disregard the administrative work of packing and unpacking tuples, the only additional work (compared to just g_2) that is being performed here, is two multiplications and an addition. This amount of additional work is the same for any step function that performs only a multiplication operation. Similarly, other arithmetic operations will also result in a constant number of arithmetic operations, meaning that the amount of additional work performed in the whole of g^{fwd} is a constant multiple of the work performed in g itself.

This holds for forward AD, but the same principle holds for reverse AD: the derivative expressions may look somewhat different (indeed, they result from vector–Jacobian multiplication, not Jacobian–vector multiplication), but the observation that the time complexity of g^{rev} is the same (up to constant factors) as for the original g will again hold.

This, then, explains why we focus on computing a single column or row of the full Jacobian: that is the part that we can compute with the same time complexity as the original function. Should we really want the full Jacobian, we can either use multiple invocations of forward AD or of reverse AD; the result (and the time complexity) is the same.

2.1.2 In practice

Essentially, the description of AD in the previous sections was on a programming language where every primitive operator takes the entire state as input and produces a new program state as output, thereby forcing any program to be a straight-line sequence of steps. While possibly instructive, most programming languages do not work like that and instead have each operation work only on a small, localised part of the program state. Indeed, g_1 and g_2 in the example above worked exactly like that.

However, the principles of AD work the same in the real world, with the same distinction between forward AD and reverse AD. Consider a slightly more complicated function than we used above:[12]

$$h = \lambda(x,y).\ x\cdot y - y + \ln(y)\cdot x$$

We could write h^{fwd} as follows, writing column vectors and row vectors equivalently as tuples and using the t_i variables for tangents of subexpressions:

$$\begin{aligned} h^{\text{fwd}} = \lambda((x,y),(xt,yt)).\ &\textbf{let}\quad (x_1,t_1) = (x\cdot y\quad ,\ y\cdot xt + x\cdot yt) \\ &\textbf{in let}\ (x_2,t_2) = (x_1 - y\ ,\ t_1 - yt) \\ &\textbf{in let}\ (x_3,t_3) = (\ln(y)\quad ,\ yt/y) \\ &\textbf{in let}\ (x_4,t_4) = (x_3\cdot x\quad ,\ x\cdot t_3 + x_3\cdot xt) \\ &\textbf{in let}\ (x_5,t_5) = (x_2 + x_4,\ t_2 + t_4) \\ &\textbf{in}\ (x_5,t_5) \end{aligned}$$

[11] The names like "xt" here are abbreviations of "x tangent", etc.

[12] This h has only a single output value. This will prevent us from illustrating again how reverse AD works with multiple output values, but since the algorithm we present in this thesis only works for functions that return a single scalar anyway, this is not a big loss.

Intuitively, this expansion can be related to the one we used for g^{fwd} earlier by intelligently eliding the parts of Jacobians that are essentially the identity matrix. Alternatively, we can explain h^{fwd} as follows: we divide execution of h up into individual arithmetic operations, and for each of those operations, we compute not only the result but also its tangent (sensitivity) with respect to an unknown value, under the assumption that xt and yt are the tangents of x and y with respect to that same unknown value. For example, writing the unknown value as ξ, knowing that $\frac{\partial y}{\partial \xi} = yt$ implies (using the chain rule[13]) that $\frac{\partial \ln(y)}{\partial \xi} = \frac{\partial \ln(y)}{\partial y} \cdot \frac{\partial y}{\partial \xi} = \frac{1}{y} \cdot yt = yt/y$, which is the value assigned to t_3 in h^{fwd}.

For reverse AD, we could write h^{rev} as follows: (note that because $h : \mathbb{R}^2 \to \mathbb{R}$, we have that $h^{\text{rev}} : (\mathbb{R}^2, \mathbb{R}) \to (\mathbb{R}, \mathbb{R}^2)$, so the second argument to h^{rev} is in $\mathbb{R}$ and thus a single scalar value[14])

$$
\begin{aligned}
h^{\text{rev}} = \lambda((x,y),d).\ &\textbf{let}\quad x_1 = x \cdot y && \vdots \\
&\textbf{in let}\ x_2 = x_1 - y && \textbf{in let}\ d_5 = d \\
&\textbf{in let}\ x_3 = \ln(y) && \textbf{in let}\ (d_2, d_4) = (d_5,\ d_5) \\
&\textbf{in let}\ x_4 = x_3 \cdot x && \textbf{in let}\ (d_3, dx^1) = (d_4 \cdot x,\ d_4 \cdot x_3) \\
&\textbf{in let}\ x_5 = x_2 + x_4 && \textbf{in let}\ dy^1 = y/d_3 \\
&\qquad\vdots && \textbf{in let}\ (d_1, dy^2) = (d_2,\ -d_2) \\
& && \textbf{in let}\ (dx^2, dy^3) = (d_1 \cdot y,\ d_1 \cdot x) \\
& && \textbf{in}\ (x_5, (dx^1 + dx^2, dy^1 + dy^2 + dy^3))
\end{aligned}
$$

As with g^{rev} in Eq. (1), the definition of h^{rev} is split in a primal part and a dual part. In the primal part we compute the value $h((x,y)) = x_5$, storing all intermediate values in variables. The indices $1, \ldots, 5$ are arbitrary, and serve just to identify the subexpressions in h.

In the dual part we use a slightly different numbering to Eq. (1) that allows a more natural intuition for what d_i means. In d_i, we store $\frac{\partial \xi}{\partial x_i}$, i.e. the partial derivative of some unknown value[15] ξ with respect to x_i, assuming that $d = \frac{\partial \xi}{\partial h((x,y))}$. This is what we called earlier the *adjoint* of x_i in the context of computing ξ; in the following, we will leave ξ implicit and talk simply about "the adjoint of x_i". Using this terminology, for the input values, the adjoint of x is stored in dx and the adjoint of y is stored in dy.

The adjoint of x_5 (the function result) is d by definition, hence the assignment $d_5 = d$ in h^{rev}. Then knowing the adjoint of $x_5 = x_2 + x_4$, we can compute the adjoints of x_2 and x_4: we get $\frac{\partial \xi}{\partial x_2} = \frac{\partial \xi}{\partial x_5} \cdot \frac{\partial x_5}{\partial x_2} = d_5 \cdot 1 = d_5$ by the chain rule, and analogously $\frac{\partial \xi}{\partial x_4} = d_5$. Therefore, we assign $d_2 = d_5$ and $d_4 = d_5$. In this way we use the adjoint of a particular subexpression x_i, and the partial derivatives of x_i with respect to the other subexpressions used in it (its "arguments", in a sense), to compute the adjoints of those arguments.

Knowing the adjoint of x_4 and x_2, we can arbitrarily choose to continue with the arguments of x_4. Since $x_4 = x_3 \cdot x$, we want to compute the adjoint of x_3 and of x—except that that x is used multiple times in h! What happens is that the real adjoint of x is the *sum* of the adjoints of all its occurrences. This is evident if we look at what happens when we vector–matrix-multiply the adjoint vector of the outputs of a function $f = \lambda(x,y).\,(x,y,y)$ with the Jacobian of that function:

$$
(d_1, d_2, d_3) \cdot Df((x,y)) = (d_1, d_2, d_3) \cdot \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 1 \end{pmatrix} = (d_1, d_2 + d_3)
$$

[13] This is yet another way of formulating the chain rule for scalar values: $\frac{\partial (g \circ f)(x)}{\partial x}(a) = \frac{\partial g(y)}{\partial y}(f(a)) \cdot \frac{\partial f(x)}{\partial x}(a)$.

[14] Strictly speaking, we should write $h^{\text{rev}} : (\mathbb{R}^2, \mathbb{R}^*) \to (\mathbb{R}, (\mathbb{R}^2)^*)$, but we again simplify using the canonical isomorphism here.

[15] This is a different ξ to the one in forward AD.

Indeed, if we have the adjoints of all the occurrences (usages) of a variable y, then the adjoint of the value of y is the sum of the adjoints of its occurrences. Each of these occurrence adjoints we call *contributions* to the adjoint of y, or *adjoint contributions* in general.

In the definition of h^{rev} above, the contributions to the adjoint of x are written dx^i for incrementing indices i; similar notation is used for y. The final adjoint dx of x is then $dx^1 + dx^2$, and the final adjoint dy of y is $dy^1 + dy^2 + dy^3$ because y is used three times in h. These sums are written in the return value of h^{rev}.

After computing d_3 (the adjoint of x_3) and dx^1, we compute the adjoint of the argument of x_3 (namely, y) in dy^1. Then, using d_2 which we computed earlier, we compute d_1 and dy^2, and finally, using d_1, we compute additional contributions dx^2 and dy^3. The return value of h^{rev} contains the normal output value and the adjoints of the arguments of h, as stated above.

Now, based on this example, we make a few observations that will be important in informing the design of our reverse AD algorithm later:

Fanout. If a subexpression has multiple usages in the function, each of those usages gets a separate *adjoint contribution*, which are summed to produce the full adjoint of that subexpression. We saw this in h where the arguments x and y were used multiple times. In other words, we can formulate this observation as saying that *fanout* in the original function translates to *addition* in the dual computation. The word "fanout" here refers to multiple outgoing edges on a node in the data-flow graph of the function. This is a common principle when discussing reverse AD on functional languages; see e.g. [42, §7]. This principle will come back in the algorithm that we present in Section 3.

Primal is necessary. Another important observation to be made here is that the dual part of the reverse AD transformation of a function really needs the intermediate values computed in the primal part. For example, in h^{rev}, the dual pass uses the value x_3 computed in the primal pass. While not all intermediate values from the primal pass are used in the dual pass, if we had none of them, we would need to recompute them each time they are used. This would compromise the important property that the time complexity of a function transformed using reverse AD is the same (up to a constant factor) as the original function, because we may end up recomputing the same value many times.

Dynamic control flow. The most important missing part in this introduction into automatic differentiation is how to handle *dynamic control flow*: with this we mean conditionals and loops. In forward AD, handling these language components is relatively straightforward: the *dual number* construction explained in Section 2.1.3 (which produces a result similar to the way we wrote h^{fwd}) extends naturally to dynamic control flow. In reverse AD, the story is more complicated because we need to *reverse* the control flow in the dual pass; for one way to handle conditionals, see the algorithm we present in Section 3. For loops, and other ways to handle dynamic control flow, refer to, for example, the implementation of reverse AD for the Julia programming language [29], tape-based approaches (e.g. [24]), or more complicated algorithms that treat the program in question in a more abstract way (see Sections 2.2.1 to 2.2.3).

Imperative languages. In this thesis we consider automatic differentiation from the perspecive of functional programming languages. Literature has also been written on the application of AD for imperative programming languages that allow mutation of variables; this is both an advantage (allowing dynamic accumulation of adjoints in the dual phase, instead of having to keep track of which adjoint contributions to add) and a disadvantage (the source program may also contain mutation). While we consider the presence of mutation in the object language out of scope in

this thesis, we will discuss in Section 2.1.3 the idea underlying (possibly in an indirect way) most implementations of reverse AD for imperative languages, which is *taping*.

Memory usage in reverse AD. The most important difference between forward AD and reverse AD is what they compute: forward AD computes a column of the Jacobian, or equivalently the tangents of the function's outputs with respect to a single input, while reverse AD computes a row of the Jacobian, or equivalently the gradient of one of the function's outputs. However, in practice, another notable difference is that reverse AD is much more resource-intensive. Even if the number of primitive operations executed is only a constant factor more than the original function would have executed, the amount of *memory* used in the process is much larger: many[16] of the intermediate values need to be kept around in memory, where the original function only had to remember all values that are actually used multiple times in the function. The administration related to this increased use of memory can play a role in making the constant factor increase in computation time between reverse AD and the original function quite significant in practice. One way to reduce the memory usage is by using the *checkpointing* optimisation as described in Section 2.1.4.

However, even if such optimisations are not applied and reverse AD is used as-is, in the case that the function in question has many input parameters and only one output parameter, computing its Jacobian (i.e. its gradient) is still much more efficient in terms of execution time using reverse AD than using forward AD. This is because reverse AD needs just one evaluation of the function, instead of one per input parameter for forward AD.

2.1.3 Implementation methods

In the discussion of AD up until now, we went from multiplication of Jacobian matrices via composition of derivative functions to a source-code transformation for a simple functional language. In the traditional AD literature, one does not go to a functional language but instead to *Wengert lists*, a representation that may be somewhat more natural when considering AD for imperative languages. A Wengert list is a sequence of *three-address operations*: $x_i = x_j \otimes x_k$ for some primitive operation $\otimes$, considering $x_1, x_2, \ldots$ to be the program state in the form of variables. Just like our exposition above, this representation does not consider dynamic control flow, and just looks at a sequence of executed operations.

The traditional way of performing AD on Wengert lists is much the same as what we did in Section 2.1.2, both for forward AD and for reverse AD. A description can be found, for example, in [35, §2.1–2.2].

If one wants to add dynamic control flow to the object language for AD, there are multiple approaches that one can take. A priori, dynamic control flow does not map nicely to the functional representation in Section 2.1.2, nor to Wengert lists: both do not (easily) support conditionally or repeatedly executing a section of the code. For forward AD, an easy way to avoid this trouble is to use the *dual number construction*. For reverse AD, the methods of solving this problem can be broadly grouped in two categories: *define-by-run* approaches and *define-then-run* approaches. We will discuss these three directions in this section.

Dual numbers. The dual number construction for forward AD can be seen as a generalisation of the method we used to write h^{fwd} in Section 2.1.2, and works very well with *operator overloading*

[16] In h^{rev} above, and in the algorithm in Section 3, the primal pass computes and stores *all* intermediate values. However, not all of these are actually needed in the dual pass. Fortunately, because our algorithm is a source-code transformation, we can benefit from dead-code elimination in an optimising compiler to remove these redundant primal values for us.

if the language in question supports this. Operator overloading is a term from certain object-oriented languages like C++, and refers to providing alternative implementations for primitive operators[17], like $+$, $-$, $\leq$, etc., when used on particular (usually library-provided) object types. Haskell has the (for our purposes) equivalent concept of *type classes* (e.g. `Num`).

The idea is that we require the to-be-differentiated program to not be written using floating-point numbers directly, but to instead be generic over the actual number type used. When this number type is instantiated to the language's concept of floating-point numbers, the function computes the normal function result. When doing forward AD, we instead instantiate this number type to a *dual number*. A dual number is a pair (x, d) of floating-point numbers, where x is the normal, primal value at that point in the computation, and d is the derivative (the tangent, see Section 2.1.1) of x with respect to some unknown value ξ, as in Section 2.1.2. Differentiable arithmetic operators and functions can be implemented for this pair type using operator overloading: $(x, d) + (y, d') = (x + y, d + d')$ since $\frac{\partial(x+y)}{\partial\xi} = \frac{\partial x}{\partial\xi} + \frac{\partial y}{\partial\xi}$; similarly, $(x, d) \cdot (y, d') = (x \cdot y, x \cdot d' + y \cdot d)$. Operations that would normally produce an integral result (such as comparison operators) discard the derivative part of the dual number: if a function f has an integral result and f is differentiable at x, then $\frac{\partial f(x)}{\partial x} = 0$, so there is no information to be kept in the dual number.

Executing a function that is generic over its number type using dual numbers, where the tangents of the input values with respect to ξ are passed in the derivative parts of the input dual numbers, produces a result that is also expressed using dual numbers. These outputs contain the tangents of the result values with respect to ξ.

In order to really obtain a high-performance forward AD implementation using the dual-numbers approach, one may need some special, additional optimisations like array-of-struct to struct-of-array translation. This puts data values of the same kind next to each other, aiding in vectorisation on CPU and GPU platforms. Some languages, typically array-oriented ones such as Accelerate, already perform such representation translations automatically, but in others additional work may be necessary.

Despite the necessity of some additional work for a high-performance implementation, implementing forward AD using dual numbers is nevertheless very natural in languages that support operator and function overloading, and can result in a very compact implementation of forward AD. However, dual numbers are not suitable for reverse AD, because they do not have any impact on the program's control flow. For reverse AD, we have to look at other methods instead, which we categorise as *define-by-run* and *define-then-run* methods.

Define-by-run. Consider a program in a hypothetical programming language that contains some dynamic control flow, and suppose we only know how to perform reverse AD on Wengert lists, which cannot represent dynamic control flow. If we execute our program on a particular input, it will take a particular path through the control flow structures contained within, executing operations along the way, and finally returning some output. An *execution trace* of the program is this list of operations executed, and this can, one way or another, be written as a Wengert list of operations. This trace will potentially only be valid for this particular input, but it is a representation that we can then perform AD on to compute its gradient for this input.

This execution trace is generally called a *tape*. Using a tape to do AD is a very runtime-centric approach: how exactly the operations on the tape will look is only determined at runtime of the program that we are trying to differentiate. In particular, when we start differentiating the Wengert list given by the operations on the tape, we are essentially writing an interpreter (that executes its input in reverse, computing the gradient) for the ad-hoc language of three-address instructions on the tape. If the programming language in question has a compiler (be it an

[17] For AD, one generally also wants other functions (`sin`, `sqrt`, etc.) overloaded for thus purpose. While this is technically function overloading and not operator overloading, we use the latter term to cover both here.

ahead-of-time compiler or a just-in-time runtime compiler), then by writing this interpreter, we are missing out on all the optimisations that the compiler could perform on the original program. To give an example: if the original program performed a multiplication that could statically be determined to be a multiplication with zero, then the compiler will probably have removed that operation from the optimised program, but our tape will still contain this operation because the compiler does not understand that this tape-write action is unnecessary too (and what to replace it with). Only if we re-implement such optimisations ourself in the differentiation transformation do we re-gain the benefits of compiler optimisations on the code being differentiated.

In addition to optimisations in the original program, an implementation of define-by-run may also miss out on optimisations that span over the whole gradient program, including both the primal and dual passes. For example, if the primal pass stores a value to memory that the dual pass never actually needs, this store can be eliminated. Detecting this in an effective manner requires insight in the source program, to be able to run a to-be-recorded analysis [23]; however, in a define-by-run implementation, the original program source is often unavailable, making such an analysis impossible.[18]

This method of writing a program execution to a tape and then differentiating that tape is called *define-by-run*: the derivative is defined by a *run* of the program. While the description above is of a naive implementation of the method, and some optimisations can be applied to improve the performance, define-by-run approaches are a priori not the most effective way to get a very fast AD implementation.

On the other hand, the advantage of define-by-run approaches is that very few restrictions need to be laid on the program to be differentiated: as long as, upon execution, the operations performed can be serialised to a tape (and they generally can be, since they have to be executed using sequential processor instructions), we can differentiate the program. This can make for easy-to-use systems that enjoy automatic compatibility with higher-level ways to write the input program.

Examples of implementations of define-by-run AD are Autograd [31] and PyTorch [41].

Define-then-run. *Define-then-run* approaches (also called *define-and-run*) are typically implemented as a source-code transformation (a term which includes AST transformations in a compiler), meaning that the code that gets compiled into machine code is not the original program, but one with additional code to compute the gradient. The key property of an implementation using define-then-run, as compared with define-by-run, is that the performed operations are not stored on a tape.

This does not mean that there must be no tape-like memory area at all: for example, Tapenade [24] implements reverse AD using a source-code transformation by storing the intermediate values, as well as bits indicating which conditional branches are taken, on a stack that is unwound by a reversed copy of the code that computes the gradient. This stack functions quite like a tape, but because the operations themselves are written as code in the second half of the transformed program, the compiler can still optimise the entire transformed program, including the gradient code.

However, to get an even larger benefit from doing AD using a source-code transformation[19], one has to do away with the tape wholesale and fine a different method of storing the primal intermediate values that is more introspectable by the compiler. This is possible for many languages [42, 55, 29], and is also what we do in the algorithm defined in Section 3. Doing this

[18] This can potentially be partially eliminated using expression templates (Section 2.1.4), but because those cannot cross dynamic control flow, it is likely they still cannot "connect" the primal and dual passes.

[19] This additionally makes it easier to express the gradient program in a purely functional language.

allows the compiler to inspect all data flow in the program, including from the primal pass to the dual pass, enabling it to optimise at will.

The difficulty of define-then-run AD is that one has to perform a non-local full-program transformation that actively rearranges control flow in the preprocessed program to run in reverse in the second pass. This difficulty depends significantly on the features in the source language: the presence of mutation, as well as complex dynamic control flow like general recursion, virtual function calls or general higher-order functions, are elements that make designing a define-then-run reverse AD transformation more difficult. On the other hand, the benefits are better optimisability of the gradient program by the compiler and lower memory usage (the operations performed need not be saved, only the necessary intermediate values). Furthermore, even aside from compiler optimisations, defining the dual pass of the gradient program as native code instead of interpreted operations on a tape already promises better performance.

A downside of a define-then-run implementation is that the code that runs is not the original code that the programmer wrote, making debugging harder (except if significant effort is made to improve this experience).

Examples of implementations of define-then-run AD are Tensorflow [1], Theano [4] and Tapenade [24].

2.1.4 Optimisations

The basic implementations of AD described above in Section 2.1.3 can be refined in multiple ways. Some well-known ways are listed here.

Retaping. If the implementation uses a tape to store the arithmetic operations and its intermediate values, for example for reverse-mode AD, then the same memory allocations, including the operations (but excluding the intermediate values), may be re-used if the same function is run again with the same inputs or if the dynamic control flow is the same. This can be used by an implementation to speculatively use the same tape as the last execution, only copying to a new tape if control flow is proven to differ [35, §3.2.2].

Checkpointing. One of the most severe downsides of naive reverse-mode AD is the necessity to record all intermediate values in the primal pass, which can cost a lot of memory. Checkpointing provides a tradeoff here between memory usage and execution time. By only storing intermediate values at certain checkpoint positions in the program, and re-running the original computation between those checkpoints when necessary, peak memory usage can be decreased at the cost of longer runtime [35, §3.2.3]. By intelligently placing checkpoints, one may also reduce the memory costs to sublinear in the program runtime, at the cost of superlinearly increasing the runtime of the Jacobian computation [50].

Expression templates. A define-by-run implementation (using taping) of reverse AD may define its programmer interface with operator overloading: the overloaded operator would, in addition to computing and returning the primal value, write the performed operation and the necessary temporary values to the tape as a side-effect.[20] As discussed in Section 2.1.3, this may result in redundant operations being written to the tape, and also in a lack of optimisations being applied to those operations.

To recover the ability to perform some compiler-like optimisations when using AD in this manner, the implementation may decide to make the overloaded operators not directly return the computed value, but instead a fragment of an abstract syntax tree (AST) corresponding to the computation.

[20] This is a viable method only in languages where invasive side-effects like these make sense, like C++.

In languages that support some compile-time computation, this allows us to perform optimisations on the collected trees that the compiler may not already be able to do, such as using a special derivative for a particular composition of functions. In general, this approach works well for sections of the program that contain no dynamic control flow, but this may depend on the metaprogramming capabilities of the language.

The AST fragments constructed by such an implementation are called *expression templates* [35, §3.3]. This technique is used in the Adept AD library [27], as well as in various libraries outside of AD, including the C++ linear algebra library Eigen[21].

Local custom derivatives. If the programmer performs a complicated computation in a function processed with AD, in particular an approximation of some transcendental function like `erf` or `lgamma` ($\ln \circ \Gamma$), there might be a simpler way to compute its derivative than to differentiate and compose all its elementary steps. In such cases, an AD implementation may allow the programmer to supply a custom Jacobian generator for such a function [35].

Supporting local custom derivatives is an important feature of performant AD implementations in practice. This is not only so that the programmer can manually provide more performant derivatives for particular pieces of code, but additionally for reasons of flexibility: local custom derivatives can also be used to *add* primitives to the source language of the AD implementation. An example could be a highly-optimised matrix multiplication routine implemented using a call to an external library, that the AD implementation cannot know how to differentiate. The derivative of matrix multiplication can probably be implemented using the same library, and with a custom derivative facility in the AD implementation, the programmer can now use this efficient matrix multiplication routine in their program to be differentiated.

Mixed-mode AD. Mixed-mode AD is a generalisation of forward/reverse-mode AD, and means not treating an entire program uniformly using reverse or forward-mode AD. It is relatively uncommon in practical AD implementations, but can be helpful in certain specialised cases, in particular when computing a full Jacobian instead of only a Jacobian–vector product. For example, if a program takes a large number of inputs, summarises those down to a single value, and then generates many outputs from that single value, the optimal strategy might be to compute the Jacobian (gradient) of the first half separately using reverse mode, then the Jacobian (elementwise derivative) of the second half separately with forward mode, and finally multiply these two (a vector–vector outer product). While computing the optimal strategy in general is NP-complete [39], heuristics can be applied, possibly with programmer aid.

2.1.5 Higher-order derivatives

Some applications, such as certain probabilistic algorithms as well as Newton's method in optimisation [35, §5.3], require second or higher-order derivatives of the function being optimised. One way to obtain these using AD is to use a sort of Taylor mode, where instead of only the first-order derivative (which is also the first Taylor series coefficient), a longer prefix of the Taylor series of the program is computed. This has been implemented in e.g. ADOL-C [22] for forward- and reverse-mode AD using operator overloading in C++. However, at least for second-order derivatives, it is usually easier to apply forward-mode AD first, followed by reverse mode over the transformed program. This computes a Hessian-vector product [35, §5.3].

The last approach can be implemented, and generalised, if the AD implementation supports nested application of the derivative operator. This is usually quite complex, especially for reverse-over-reverse, and can have subtle implications on correctness and usability, depending in severity on the strengh of the type system in the host language [34].

[21] For example, see http://eigen.tuxfamily.org/index.php?title=Expression_templates.

2.2 Functional AD

The discussion of implementation methods in Section 2.1.3 mostly concerned the traditional context of AD, which is impure imperative programming languages. For functional languages, especially pure ones, literature proposes a number of ways to implement reverse-mode AD. Forward mode is seen as a solved problem in this section, since the operator overloading implementation with dual numbers is completely satisfactory: it can be implemented as a library, and all computations are transparent to the compiler, allowing the compiler to optimise the code at will. A good example is the implementation in [49], which applies further optimisations to forward AD to make it feasible even for some gradient-computation tasks. The only real difficulty is computing higher-order derivatives using iterated AD, in which case one must watch for perturbation confusion [34].

2.2.1 On untyped lambda calculus

In a seminal paper [42], an operator on a Scheme-like untyped lambda calculus is described that can be used to perform reverse AD on any expression in the language. This AD operator, $\overleftarrow{\mathcal{J}}$, also supports self-application, so that it can be used to support *nested* reverse AD for higher-order derivatives or nested optimisation. The operator is fairly complex; this is due to the complexity of the language that it works on (being both untyped and higher-order), as well as due to the support for nested AD.

An AD transformation on a language L is allowed to produce output that uses the full language L. The operator $\overleftarrow{\mathcal{J}}$ uses this freedom: even if the input program does not contain any higher-order structures, the output will contain many inline closures that are treated as first-class objects. Accelerate is not a higher-order language, so using the operator $\overleftarrow{\mathcal{J}}$ to implement an AD transformation on Accelerate would not be possible without at least some amount of effort to eliminate the usage of higher-order structures in its output. Furthermore, while the generality of the input language should allow translation of Accelerate's array primitives to something that $\overleftarrow{\mathcal{J}}$ understands, it is unclear whether the output can always be converted back again to array primitives. To avoid these translation issues between untyped lambda calculus and the Accelerate language, and possibly define a simpler AD operator (for our simpler language) along the way, we choose to not directly apply the methods from [42] on Accelerate in this thesis.

2.2.2 Delimited continuations

If the host language has built-in support for the shift/reset control operators (implementing delimited continuations [13]), there exists a very natural reverse-mode AD source code transformation in terms of shift/reset and some mutable cells [58]. While shift and reset can be eliminated using a continuation-passing style (CPS) transform [45], the resulting code is not in standard CPS form: calls to continuations are not always tail calls. This prevents the standard un-CPS transformation [19] from applying here.

An alternative method to reduce the code back to a first-order program is using defunctionalisation [46]. This is also more generally applicable on code that uses function calls for control flow instead of structures like sequencing and loops. The idea is to create a sum type, with one constructor for each creation of a lambda function in the program. The arguments of those constructors are the free variables of the corresponding lambda functions. Introductions of lambda functions are then replaced with invocations of the corresponding sum type constructors, and calls of function-typed variables are replaced with a call to `apply`: this is the elimination function for lambda tags, and takes a sum type arm and an argument, then executes the code in the body of the original lambda function for that argument. For a survey and more thorough treatment of defunctionalisation, see [14].

Aside from the usage of control operators, an additional complication using the AD method

from [58] is that the produced program contains mutation. This mutation is limited to additive accumulators, but since our language (Accelerate) is pure, this nevertheless needs to be emulated, or statically eliminated. This is a possible future research direction.

2.2.3 Via categorical language

The simply-typed lambda calculus (STLC) can be modelled by any Cartesian Closed Category (CCC); for our purposes, a CCC can be seen as a Haskell data type that is an instance of a particular type class `CCC`. The operations defined by `CCC` are such that it can express STLC programs in a combinator representation, or point-free style [17]. This combinator representation has number of properties that make it a natural language to express AD in [18, 55]; this is one of the possible interpretations of this combinator form, with others including one corresponding to normal evaluation via β-reductions [17].

The result of the AD interpretation of `CCC` (at least if one wants explicit control over the AD mode, e.g. forward or reverse) is point-free code in something resembling continuation-passing style that computes both the primal and the derivative result. There is no explicit tape, graph or other such structure, so storage is implicit in closures. It is currently unclear how to compile this code down to an efficient first-order program that can be run on actual hardware, but progress is being made [55].

2.2.4 Via compiler optimisations

A different way to approach reverse-mode AD is to start with forward mode and try to optimise it to something resembling reverse mode. This has been shown [49] to work on a functional array language based on pull-arrays: the only array-construction operation is `build`, corresponding to `generate` in Accelerate, and the only array-consumption operation is `ifold`, corresponding to `fold` $\circ$ `indexed` in Accelerate. Forward-mode AD is defined using the dual numbers approach; then a number of compiler optimisations can be defined [49, Fig. 8] that use distributivity laws in various well-chosen places to simplify out the duplicate work in computing a gradient using forward-mode AD.

If the original program does not contain significant dynamic control flow, this can produce the efficiency of reverse mode while not actually implementing reverse mode directly. While recent work [10] has proven that the transformations in [49] are semantically correct, it is yet unclear for which input language it can be guaranteed that these compiler optimisations really produce a result with the time complexity of reverse AD.

This approach to reverse mode is attractive because it solely consists of local transformations and does not involve mutation, nor higher-level structures like closures, continuation-passing style, program reinterpretations, etc. However, since dynamic control flow cannot be easily handled by these distributivity-based rewritings, the method would at least need to be extended before it can be applied effectively on more complex programs.

2.3 Functional programming

Besides automatic differentiation, the topic of this thesis also touches on some areas of functional programming research, most particularly functional array languages and type-safe program representations. Accelerate is a compiler for a functional array language, and during the compilation process it represents the subject program in a type-safe AST, using generalised algebraic data types (GADTs) and De Bruijn-indexed environments. We will cover these topics in the following subsections.

2.3.1 Functional array languages

The creation of functional array languages is driven by the wish for a language in which one can effectively express aggregate, parallelisable computations, with the condition that it must be easy to reason about its semantics. This latter point pertains both to software reasoning as well as human reasoning: for software it means opportunities for optimisation, compilation for various types of backends, and possibly correctness analyses; for humans, on the other hand, it means a more direct interface to that software, and more insight into correctness of the program. The focus on aggregate, parallelisable computations is to be able to exploit massively parallel hardware like GPUs, which are the typical backend for performance-oriented array languages [8, 25, 52, 28].

The trend for functional array languages is to build on a certain set of higher-order primitives for aggregate computation. The core of this set consists of `generate`, which takes an array size specification and a function from array indices to values, and produces an array filled with the outputs of that function; and `fold`, which uses an associative binary combination function to reduce, or summarise, one or more dimensions of an array to a single value. This way of basing the array construction primitives on `generate` (also called "build" in other settings) is the core of *pull arrays* [53]; there are alternatives.

This basic language can be extended, either by enlarging the set of possible computations the language can efficiently express, or by implementing special-purpose operators that—while functionally subsumed by existing operators—can be implemented more efficiently than their more general counterpart. An example of the first type is `scan`, which computes the prefix sums of an array, where the +-operator may be replaced by any associative binary operator. `scan` has been shown to have significant applications as a highly-optimised basic operator in imperative code [9], and can also be efficiently implemented on GPUs [48]. The functionality exposed by `scan` cannot be obtained with reasonable time complexity using `generate` and `fold`. A different enlarging example is *segmented* versions of `fold` and `scan`: these perform their respective operation on subarrays demarcated by a segment array (or bitvector, depending on the implementation). All these operators are implemented in Accelerate.

Examples of the second kind (specialisation of a more general primitive) include `map` and `zip`. Another example is given by an operator that Accelerate does not currently implement specially, but that does exist in e.g. Futhark: `fold_comm`[22] is a version of `fold` that assumes its binary operator is not only associative, but also commutative. This allows a more efficient parallel implementation of the fold.

An additional specialisation example is a primitive for stencil operations. Stencil operations are somewhat like `map`, except that in the computation of an output element, the computation may use not only the corresponding element in the input array but all elements in a small window (e.g. of radius 1 or 2) around that input element. They are used for various kinds of physical simulations, and have been the focus of research regarding their efficient implementation on GPUs [47], also in Accelerate [21]. Stencils are special cases of `generate`, but have more advanced specialised algorithms available, capitalising on the spatial locality of their memory access pattern.

2.3.2 The Accelerate language

The Accelerate language is an embedded domain-specific language (EDSL) in Haskell. This means that a program written in Accelerate is a piece of Haskell code that is written using (almost) only functions and types provided by the `accelerate` [12] library. Any operations from outside this library that the program uses are in essence metaprogramming from the perspective of Accelerate: they determine how the elements of the EDSL are combined to form the program in the real Accelerate language.

[22] Called `reduce_comm` in Futhark: https://futhark-lang.org/docs/prelude/doc/prelude/soacs.html.

As a consequence, the Accelerate language essentially consists of a number of primitives, implemented in, and exported by, the `accelerate` library. The following is a (small) example of an Accelerate program:

```
\a -> fold (+) 0 (map (\x -> x * x) a)
```

When used with the `accelerate` library (which overrides a number of definitions from the standard Haskell prelude), `fold` and `map` in the above code fragment have the following types:

```
fold :: (Elt a, Shape sh) => (Exp a -> Exp a -> Exp a) -> Exp a -> Acc (Array (sh :. Int) a)
     -> Acc (Array sh a)
map :: (Elt a, Elt b, Shape sh) => (Exp a -> Exp b) -> Acc (Array sh a) -> Acc (Array sh b)
```

The type of `fold` implies that the operator `(+)` in the above fragment gets, for some admissible Accelerate element type `a`, the type `Exp a -> Exp a -> Exp a`, and indeed for the expected values of `a`, `Exp a` is an instance of the standard Haskell `Num` type class.

An array in Accelerate has type `Array sh a`, where `a` is the type of the elements of the array (e.g. `Float`), and `sh` is the *shape type* of the array. An array's shape type encodes the *rank* (the number of dimensions) of the array on the type level. An example of a shape type in the Accelerate EDSL is `Z :. Int :. Int`, which is the shape type for a two-dimensional array. `Z` is the shape type for a zero-dimensional array, containing exactly one element. The operator `(:.)` associates to the left, so if `sh` is the shape type of an n-dimensional array, `sh :. Int` is the shape type of an $(n+1)$-dimensional array. A two-dimensional array with elements of type `Float` has type `Array (Z :. Int :. Int) Float`; using some type aliases this may alternatively be written `Array DIM2 Float` or even `Matrix Float`. The `Shape` type class contains exactly the types `Z`, `Z :. Int`, `Z :. Int :. Int`, etc.

Ignoring for a moment the "`Exp`" and "`Acc`" occurring in the types, we can now explain what these primitives, and thus the program shown above, do. The `map` primitive constructs a new array containing as values the results of applying its first argument to each element of the array in its second argument. The `fold` primitive takes an associative combination function and an initial value, and reduces along the inner dimension of the array argument (placing the initial value on the left). The result of a `fold` is an array with one dimension less than the argument array: the inner dimension has been summarised into a single value.

Therefore, the program `(\a -> fold (+) 0 (map (\x -> x * x) a))` computes the sum of squares along the inner dimension of its argument array.[23] If `a` is a single-dimensional array, the result is a zero-dimensional array containing a single value; if `a` is a two-dimensional array, the result is a one-dimensional array containing the sum-of-squares results of the rows[24] of `a`. The `map` and `fold` primitives are *rank-polymorphic*, and as a result, our example program is too.

Staged compilation. We said that an array in Accelerate has type `Array sh a`, but the inferred type of our example program above by the Haskell compiler is the following:[25]

```
(Num a, Shape sh) => Acc (Array (sh :. Int) a) -> Acc (Array sh a)
```

We cannot simply pass an array to this function, because we would need to somehow lift our array to something within `Acc`. So what is this `Acc`?

[23] Should one wish to reduce over the entire array instead of only the inner dimension, one can replace `fold` with `foldAll`, which returns a value of type `Acc (Array Z a)`, a zero-dimensional array which contains one element. `foldAll` is defined in terms of the `reshape` and `fold` primitives, by first flattening the argument to a single-dimensional array and then applying `fold`.

[24] That is, if a two-dimensional array is seen as being in row-major order. This is just a question of interpretation; Accelerate itself is well-defined speaking just about the "inner dimension".

[25] The `Num` type class here is Accelerate's version, defined as `type Num a = (Elt a, Prelude.Num (Exp a))`.

Accelerate is an embedded language, and the real compilation of the Accelerate program does not happen until runtime of the Haskell program: the Accelerate compiler runs at runtime of the Haskell code (see Section 2.3.4).[26] Therefore, all the primitives themselves actually do is build up an *abstract syntax tree* (AST) of the program in the Accelerate language. A value of type `Acc a` is the AST of an Accelerate program that would produce a value of type `a` when evaluated, for all `a` that are (possibly nested) tuples of `Array sh t` types. For scalar (i.e. non-array) types `a`, Accelerate uses `Exp a` instead.

Evaluation of an Accelerate program is done using the `run` function provided by an Accelerate backend, which has type `Arrays a => Acc a -> a`; the `Arrays` type class specifies that `a` must be a tuple of `Array sh t` types. This `run` function compiles and then evaluates the program; in practice, one should probably instead use the `runN` function that allows compiling a program once and running it many times on different inputs. The Accelerate documentation contains more information on the specifics here.

Conversely, to lift constant values from Haskell into an Accelerate program, the primitives `constant :: Elt a => a -> Exp a` and `use :: Arrays a => a -> Acc a` are provided.

Stratification. In the Accelerate compiler, the Accelerate language is stratified in two separate languages: the *array sublanguage* and the *expression sublanguage*. In the embedded language in Haskell that Accelerate presents to the user, these correspond respectively to the primitives returning `Acc` values and those returning `Exp` values. The array sublanguage contains operations on arrays like `generate`, `map`, `fold`, `scan`, `permute`; these are the aggregate array operations that make Accelerate a parallel array language. The arguments to these primitives are either arrays (which can, of course, be the result of another array primitive), or expressions or expression functions. An expression function is no more than a Haskell function that takes arguments in `Exp` and returns a result in `Exp`; a program represented by an AST in `Exp` is called an *expression*. The expression sublanguage provides all of the expected arithmetic operations for integers, floating-point numbers, etc. and is used to write the scalar computations that are performed in parallel in the aggregate array operations.

In addition to these basic operations, both the array and the expression sublanguage allow for constructing and projecting tuples of values, as well as for conditionals (if-expressions) and unbounded loops (while-loops). Furthermore, the expression sublanguage contains primitives for querying the *shape* of an array, as well as for *indexing* into an array.

The shape of an array of type `Array sh a` is a value of type `sh`, hence the `shape` function in the EDSL has type `(Shape sh, Elt a) => Acc (Array sh a) -> Exp sh`. Recall that shape types are of the form `Z` or `sh :. Int` for a shape type `sh`; since these types are defined as follows:

```
1 data Z = Z
2 data sh :. i = sh :. i
```

the shape of a two-dimensional array can be e.g. `Z :. 2 :. 3` for a 2-by-3 array.

The array indexing operator has the following type:

```
1 (!) :: (Elt a, Shape sh) => Acc (Array sh a) -> Exp sh -> Exp a
```

so that, for example, the top-left element in a two-dimensional array `a` can be obtained using the code `a ! constant (Z :. 0 :. 0)`.[27]

[26] Actually, Accelerate supports running EDSL compilation at compile-time of the Haskell program using TemplateHaskell. We glance over that in the discussion here.

[27] The examples here do not use non-constant array indices, but this is not because of any inability in Accelerate: Accelerate provides a pattern synonym `::.` that allows transparently converting between shape-typed values and lifted versions of those values. For example, if `i` and `j` have type `Exp Int`, then `Z ::. i ::. j` has type `Exp (Z :. Int :. Int)`, whereas `Z :. i :. j` would have type `Z :. Exp Int :. Exp Int`, which is not a useful

A crucial characteristic of the `shape` and `(!)` primitives in the expression sublanguage is that they take an `Acc` value as argument, but return an `Exp` value. This has the potential for producing *nested parallelism* [8]; consider for example the following three programs, all of which have type `Num a => Acc (Array (Z :. Int) a) -> Acc (Array (Z :. Int) a)`:

```
1  \a -> let b = sum (map (+ 3) a) in map (\x -> x + b ! constant Z) a
2  \a -> map (\x -> x + sum (map (+ 3) a) ! constant Z) a
3  \a -> map (\x -> x + sum (map (+ x) a) ! constant Z) a
```

The first two are exactly equivalent[28]: first the sum is computed over the values of `a` with 3 added to each element, and then we return an array that is `a` with the just-computed sum added to each element. These are fine Accelerate programs. The third program, however, is different in that the first parallel computation, `sum (map (+ x) a)`, depends on the value `x` and thus cannot be lifted out with a let-binding.

The third program exhibits nested parallelism: the individual computations in the outer parallel `map` operation themselves also use parallel operations to compute their result. Accelerate does not currently allow this, and rejects the third program at runtime, when the staged Accelerate program is analysed and compiled to native code. Unfortunately, nested parallelism is not yet detected and forbidden by the type system at compile time of the Haskell code.

The prohibition of nested parallelism ensures that the stratification of Accelerate into an array sublanguage (which contains parallel operations) and an expression sublanguage (which contains only non-parallel scalar operations) cannot be cheated. Furthermore, since the elements of an array can only be scalar types in Accelerate, arrays are necessarily "rectangular", meaning that all parallelism in Accelerate is *regular parallism* [33].

Second-order representation. The stratification into a parallel array sublanguage and a sequential expression language makes Accelerate not a higher-order language, but a *second-order language*. While morally the aggregate array operations described in Section 2.3.1 are higher-order, in Accelerate they are restricted by the prohibition of nested parallelism, thus creating only a single "layer" above the first-order expression language.

This separation, of the language of parallel operations and the language of the sequential scalar operations within those parallel operations, serves to make efficient compilation into GPU kernels easier. Indeed, a GPU kernel consists of a piece of code running on a single, sequential GPU thread, and a specification for running that piece of code in parallel over all the cores of the device. In CUDA[29], this sequential piece of code is the `__kernel__` function, and the parallel specification corresponds with the `<<<...>>>` call syntax.

A thread in a GPU kernel cannot start new GPU kernels; an exception to this restriction is device memory allocation, but even this is suboptimal from a performance standpoint. Therefore, compiling for a GPU means that, at least at the end of the compilation pipeline, one cannot have any nested parallism in the program to be compiled. Therefore, to support nested parallism in the EDSL, the Accelerate compiler would have to convert the program to one or more separate programs that are each devoid of nested parallism. Performing this conversion in a general way that also results in performant code, is not an easy problem.

More recent work on Accelerate has implemented support for a limited amount of irregular nested parallelism in the form of sequences of array computations: [33] implements regular sequences for dynamic work distribution, which a later publication [11] extends to irregular sequences.

type in Accelerate (except to lift to the former type). The pattern synonym is bidirectional and can therefore also be used for binding variables, in which case it unlifts the `:.` elements of the value and leaves the indices lifted.

[28] In fact, the Accelerate compiler cannot distinguish between them, because at that time `sum (map (+ 3) a)` has already been inlined.

[29] https://developer.nvidia.com/cuda-zone

However, this functionality is not currently exposed in a released version of Accelerate, and the differentiation algorithm presented in this thesis supports only the regular subset of Accelerate.

In addition to being useful for compilation to GPU kernels, the second-order form of the language is helpful for designing an algorithm for reverse automatic differentiation that works on Accelerate. Performing reverse-mode AD on a full higher-order language is still a hard problem: constructions like the one discussed in Section 2.2.1 exist, but are very complex and not easily implemented in a restricted language. However, performing reverse AD on a second-order language is much more doable, as evidenced by the algorithm presented in this thesis.

2.3.3 Type-safe program representations

Accelerate is an embedded domain-specific language (EDSL) in Haskell, and thus borrows its metaprogramming capabilities. For such a language, an important choice with regards to its embedding is whether to denote name-binding constructs, i.e. lambda functions and let-bindings, using the Haskell-native constructs (corresponding to higher-order abstract syntax (HOAS) [43]) or with custom primitives (corresponding to an inductive term representation) [2]. An inductive representation is well-suited for common compiler transformations, but using it for the embedded language interface results in a very inconvenient language for the programmer. The HOAS representation results in a simpler and more familiar language for the programmer (since they can use the Haskell-native constructs for name binding as usual), but is less easy to operate on for a compiler, because portions of the program tree are hidden behind actual closures and there is sharing in the tree that a compiler can easily unwittingly destroy.

A solution is to use a HOAS-based embedding, then convert to a usual inductive representation for internal use in the compiler; this is what the Accelerate compiler does in its sharing recovery pass [37].

A largely orthogonal property of a program representation is whether it is typed: in a Haskell implementation, this corresponds to whether GADTs are used for the AST, as opposed to regular sum types. In the typed case, the internal representation of a subterm in the compiler has a type that contains (a representation of) the type of the subterm in the language being compiled. This way of using GADTs for representing expressions is in fact one of the standard, and original, motivating examples for GADTs in functional languages [3]. For a good example of using GADTs for representing expressions, and a description of their implementation in the GHC Haskell compiler, see [56].

The advantage of using a typed representation[30] is that any program transformation algorithm that typechecks in the language the compiler is implemented in, performs a type-correct transformation on the program being compiled. Even though this does not imply that the transformation is also semantically correct, it does give an extra measure of confidence, especially if program subterms have (partially) generic types for which parametricity arguments (see e.g. [57]) can be invoked.

In a typed representation, variable references need to be typed as well; however, adding a type tag to normal named variable references does little to enforce program type-correctness (since the usage of a variable is not linked with its definition). To link use and definition, the environment of variables currently in scope needs to be available at the point of a variable reference. While this can be done with a dependent map from the variable name to its definition[31], one can actually remove all names (and make alpha-equivalence trivial along the way) by switching to De Bruijn indices [15]. This allows the environment to be a stack of types and variable references

[30] More specifically, this is sometimes called *intrinsically well-typed* De Bruijn style [6], to differentiate it from an *extrinsic* representation that keeps the evidence of type-correctness of the AST separate from the AST itself.

[31] For example, using the Haskell `https://hackage.haskell.org/package/dependent-map` library, or more naturally in a dependently-typed language.

to be Peano-natural-based indices into this type-level list, and because the same position in the environment is linked via types to both the definition and the use site, type-correctness of the AST implies type-correct variable references [2].

This typed representation using GADTs and typed De Bruijn indices is precisely the AST representation in the Accelerate compiler.

2.3.4 The Accelerate compiler

The Accelerate compiler is written in Haskell, and compiles an EDSL in Haskell (called the Accelerate language, or also Accelerate for short) to machine code for a number of target platforms. The array program is written by the programmer in their Haskell program, which is then compiled using the GHC Haskell compiler to an executable. At runtime of that executable, the array program evaluates to a representation of that program using HOAS, which is then passed to the Accelerate compiler. This is an example of staged compilation (see e.g. [58]).

Now in the Accelerate compiler, the program is first converted to an inductive representation with typed environments and De Bruijn indexing using the previously mentioned *sharing recovery* pass. Aside from eliminating the HOAS structures, this pass also detects sharing (possibly introduced by GHC's common subexpression elimination optimisation, or user-written let-bindings) and converts that to explicit let-bindings in the AST [37]. The type of the resulting inductive representation is a GADT which models the full Accelerate language, and is also called the main internal AST. This representation is the one on which other program transformations can be most naturally applied, such as vectorisation [33].

Afterwards, the *array fusion* pass matches producers, transducers and consumers together to (attempt to) fuse multiple aggregate array operations into fewer operations, to reduce execution overhead and yield a more efficient array program [37]. This results in a variant of the AST that includes *delayed* array representations; this delayed representation is really a variant of pull-arrays (representing all arrays using `generate`-like expressions) that makes a distinction between certain classes of generate-expressions to support more specific optimisations.[32]

After fusion is complete, a backend like `accelerate-llvm` takes over and handles the rest of the way towards high-performance machine code on whatever hardware is currently targeted.

2.3.5 Automatic differentiation

The most suitable place to implement a reverse-mode AD transformation in the Accelerate compiler pipeline is after the sharing recovery pass, when the program is represented in an inductive representation as the main internal AST. Choosing this location means that the transformation does not have to preserve implicit sharing in the AST (since that has already been made explicit in the sharing recovery pass), but can still benefit from the array fusion optimisations. The fact that these existing optimisations can still run on the generated derivative code is precisely the reason why implementing reverse-mode AD using a source-code transformation is expected to give a high-performance result.

2.3.6 Typed AST

As mentioned, the Accelerate compiler uses the GADT with typed De Bruijn environments approach described above for its program representation (at least, after sharing recovery). This ensures that when compiling the Accelerate compiler, type-correctness of program transformations is verified using the types on the AST. However, GHC's typechecker is not always intelligent

[32] Essentially, this is another example of language expansion via specialisation as described in Section 2.3.1, except on the internal language instead of the user-facing language.

enough to be able to decide type-correctness of all transformations; in particular, this is the case for one point in the sharing recovery pass [37, §3]. Here, the Accelerate compiler uses reified types to perform the type checking at runtime; this then produces a type equality witness that tells GHC that the transformation is indeed correct [36, §5.1.1]. This particular instance of runtime type checking has also been independently indicated as being probably necessary in the context of general HOAS conversion [2, §2.6].

It is to be expected that such runtime type checking is necessary for more kinds of program transformations that are not strictly local in nature. In particular, since a reverse-mode AD transformation produces a program that reverses the control flow of the original program, it would be remarkable if that can be implemented without any runtime type checking. Indeed, as we will see in Section 4.2.4, some runtime type checking was needed for our implementation of the algorithm in Section 3 in Accelerate.

3 The AD algorithm

In this section, we will explain the algorithm used to perform reverse-mode automatic differentiation on Accelerate programs in our implementation. In order to be able to present the algorithm more clearly, and to avoid requiring Accelerate-specific knowledge only to understand the AD algorithm, we will use a model of the language called Idealised Accelerate. We list the important differences between Idealised Accelerate and the actual Accelerate language in Section 3.2.1.

Since Idealised Accelerate, like the Accelerate itself, is stratified into an array sublanguage and an expression sublanguage, the AD algorithm presented also consists of two halves. Both halves of the algorithm look very much alike, but due to the interaction between the array and expression sublanguages, the two halves of the algorithm are necessarily interlinked.

The object language, Idealised Accelerate, will be described in Section 3.1. We will give a specification of our algorithm in Section 3.2, and discuss some examples in Section 3.3. Then, we will separately discuss the two halves of the algorithm: first we will look at the array AD algorithm in Section 3.4, and then at how the expression AD algorithm differs in Section 3.5.

3.1 Idealised Accelerate

Idealised Accelerate is a pure functional language for array processing: mutation is not observable in the language. A basic grammar of Idealised Accelerate can be found in Fig. 1. Like Accelerate, this language is stratified into two sublanguages: the expression sublanguage (*exp*) and the array sublanguage (*arr*). There are also expression functions (*fun*) and array functions (*afun*); here we allow only single-argument functions, because it makes the analysis easier and because multi-argument functions can be easily uncurried to their single-argument equivalents. Expression variables (*evar*) and array variables (*avar*) are here specified to only take the form x_i and a_i, respectively, but in practice we will be more liberal with variable names to aid readability. The meaning will always be clear in context. Primitive expression operations (*op*), which are used in the `PrimApp` expression constructor, have a few representative examples listed in the figure, but more could be easily added.

The stratification of the language into two layers (*arr* and *exp*) makes the language second-order: array-level code contains functions with expression-level code, but expression-level code does not contain any higher-order constructions. Not only does this help in producing efficient GPU code in the Accelerate compiler (as mentioned before in Section 2.3.2), this also makes designing an AD algorithm more tractable.

The language has a simple type system. Types are considered to be inferred; if the type of an expression is unclear or ambiguous, we will use superscript type annotations. The implementation of a typechecker and the accompanying more ergonomic user-facing language is orthogonal to the AD algorithm, and thus considered out of scope.

All expressions (produced by *exp*) have a *scalar type* (written t in Fig. 1), which is either the empty tuple $()$, a *single-type*, or a pair of scalar types (t_1, t_2). A single-type is either a floating-point type (f); an integral type (i), for example `int`; or the boolean type `bool`. Exactly which types are offered is immaterial for the discussion, as long as they can be divided into the category of "continuous types" and "discrete types" with the property that a differentiable function that returns a discrete type must be locally constant and thus have derivative 0. Function types are written in the usual way with $\to$.

The type of an array with element type t and *index type* s we denote as $t[s]$. The index type of an array is $()$ for a zero-dimensional array, $((), \texttt{int})$ for a one-dimensional array, $(((), \texttt{int}), \texttt{int})$ for a two-dimensional array, etc. For example, to retrieve the element of a two-dimensional array `a` indicated in C-style syntax as `a[2][3]`, the following expression can

exp	$::=$ $\texttt{Const}^{t}$ (*constant literal of type* t)	constant introduction
	$\mid$ $\texttt{Evar}^{t}$ $evar^{t}$	expression variable reference
	$\mid$ $\texttt{Let}^{t'}$ $evar = exp^{t}$ $\texttt{in}$ $exp^{t'}$	let-binding
	$\mid$ $\texttt{Nil}^{()}$	empty tuple
	$\mid$ $\texttt{Pair}^{(t_1,t_2)}$ exp^{t_1} exp^{t_2}	pair
	$\mid$ $\texttt{Fst}^{t_1}$ $exp^{(t_1,t_2)}$	left pair projection
	$\mid$ $\texttt{Snd}^{t_2}$ $exp^{(t_1,t_2)}$	right pair projection
	$\mid$ $\texttt{PrimApp}^{t'}$ $op^{t\to t'}$ exp^{t}	primitive expression operation
	$\mid$ $\texttt{Index}^{t}$ $avar^{t[s]}$ exp^{s}	array indexing
	$\mid$ $\texttt{Shape}^{s}$ $avar^{t[s]}$	array shape query
	$\mid$ $\texttt{Cond}^{t}$ $exp^{\texttt{bool}}$ exp^{t} exp^{t}	expression-level conditional
fun	$::=$ $\texttt{Lam}^{t\to t'}$ $evar^{t}$ $exp^{t'}$	expression function abstraction
arr	$::=$ $\texttt{Use}^{t[s]}$ (*array literal of type* $t[s]$)	constant array introduction
	$\mid$ $\texttt{Avar}^{T}$ $avar^{T}$	array variable reference
	$\mid$ $\texttt{Alet}^{T}$ $avar^{T'} = arr^{T'}$ $\texttt{in}$ arr^{T}	let-binding
	$\mid$ $\texttt{Anil}^{()}$	empty tuple
	$\mid$ $\texttt{Apair}^{(T_1,T_2)}$ arr^{T_1} arr^{T_2}	pair
	$\mid$ $\texttt{Afst}^{T_1}$ $arr^{(T_1,T_2)}$	left pair projection
	$\mid$ $\texttt{Asnd}^{T_2}$ $arr^{(T_1,T_2)}$	right pair projection
	$\mid$ $\texttt{Reshape}^{t[s']}$ $exp^{s'}$ $arr^{t[s]}$	array reshape
	$\mid$ $\texttt{Replicate}^{t[s']}$ $exp^{\mathrm{Slc}[s\rightsquigarrow s']}$ $arr^{t[s]}$	dimension replication
	$\mid$ $\texttt{Slice}^{t[s]}$ $exp^{\mathrm{Slc}[s\rightsquigarrow s']}$ $arr^{t[s']}$	dimension elimination
	$\mid$ $\texttt{Generate}^{t[s]}$ exp^{s} $fun^{s\to t}$	generic array builder
	$\mid$ $\texttt{Map}^{t'[s]}$ $fun^{t\to t'}$ $arr^{t[s]}$	element-wise operation
	$\mid$ $\texttt{ZipWith}^{t[s]}$ $fun^{(t_1,t_2)\to t}$ $arr^{t_1[s]}$ $arr^{t_2[s]}$	two-input elementwise operation
	$\mid$ $\texttt{Sum}^{t[s]}$ $arr^{t[(s,\texttt{int})]}$	specialised fold over inner dimension
	$\mid$ $\texttt{Foldl}^{t[s]}$ $fun^{(t,t)\to t}$ exp^{t} $arr^{t[(s,\texttt{int})]}$	fold over inner dimension
	$\mid$ $\texttt{Scanl}^{t[s]}$ $fun^{(t,t)\to t}$ exp^{t} $arr^{t[s]}$	inner-dimension left-scan
	$\mid$ $\texttt{Scanr}^{t[s]}$ $fun^{(t,t)\to t}$ exp^{t} $arr^{t[s]}$	inner-dimension right-scan
	$\mid$ $\texttt{Permute}^{t[s']}$ $fun^{(t,t)\to t}$ $arr^{t[s']}$ $fun^{s\to(\texttt{bool},s')}$ $arr^{t[s]}$	forward permutation (scatter)
	$\mid$ $\texttt{Backpermute}^{t[s']}$ $exp^{s'}$ $fun^{s'\to s}$ $arr^{t[s]}$	backward permutation (gather)
	$\mid$ $\texttt{Acond}^{t[s]}$ $exp^{\texttt{bool}}$ $arr^{t[s]}$ $arr^{t[s]}$	array-level conditional
$afun$	$::=$ $\texttt{Alam}^{t\to t'}$ $avar^{t}$ $arr^{t'}$	array functions
$evar$	$::=$ $x_1 \mid x_2 \mid \ldots$	expression variables
$avar$	$::=$ $a_1 \mid a_2 \mid \ldots$	array variables
op	$::=$ $\texttt{Add}^{(t,t)\to t} \mid \texttt{Mul}^{(t,t)\to t} \mid \texttt{Lt}^{(t,t)\to\texttt{bool}}$	primitive expression operations
	$\mid$ $\texttt{ToFloating}^{i\to f} \mid \texttt{Round}^{f\to i} \mid \ldots$	

Figure 1: Idealised Accelerate, an Accelerate-like language that we define the AD algorithm on in Section 3. t, t', t_1, t_2 are arbitrary scalar types; s is an index type; i is an integral type; f is a floating-point type; $t[s]$ is an array of ts with shape s; T, T', T_1, T_2 are arbitrary array tuple types; $\mathrm{Slc}[s \rightsquigarrow s']$ is a slice with small shape s and large shape s'.
See Section 3.1.

be used: `Index a (Pair (Pair Nil (Const 2)) (Const 3))`. The `Shape` constructor of *exp* returns the shape of the given array variable, where the components indicate not an index into the array but the size of the corresponding dimension. For example, if the array `a` has size 10×20, the expression `Shape a` will have value $(((), 10), 20)$.

All array programs (produced by *arr*) have an *array tuple type*, written T, which is either the empty tuple $()$, an array type $t[s]$, or a pair of array tuple types (T_1, T_2). Most *arr* constructors, however, return and take individual arrays. The notation $\mathrm{Slc}[s \rightsquigarrow s']$ is used for `Replicate` and `Slice`, and stands for a *slice type* with small shape s and large shape s'. A slice type is recursively defined as either $()$, $(slc, \texttt{int})$ or $(slc, ())$ for any slice type *slc*. The small and large shapes of a slice type are best explained using an example: the slice type $((((((), \texttt{int}), ()), ()), \texttt{int}), \texttt{int})$ has small type $(((), \texttt{int}), \texttt{int})$ (an "`int`" for each $()$ in the slice type) and large type $((((((), \texttt{int}), \texttt{int}), \texttt{int}), \texttt{int}), \texttt{int})$ (an "`int`" for all entries in the slice type). For concrete examples of slice-typed values, see the informal semantics of `Replicate` and `Slice` in the list below.

We now give an informal (denotational) semantics of the less evident language elements. Note that while the array operations are phrased using mutable language terminology, the language is pure; hence, the descriptions should be understood to describe the output in terms of the input.

- `Const` and `Use` introduce scalar and array constants, respectively. These are not values that we will differentiate with respect to. Note that `Const` can also introduce values like $(1, \texttt{true})$ of type $(\texttt{int}, \texttt{bool})$.
- `Index` and `Shape` are as described above.
- Conditionals (`Cond` and `Acond`) return their second or third argument, depending on the first (condition) argument. The branch not taken is not executed; the conditional primitives are the only ones that are not fully strict.
- `ZipWith` functions as one would expect, but we want to emphasise its behaviour when the arguments have different shapes (while the number of dimensions must be the same, the actual sizes of the two argument arrays may differ). The result of a `ZipWith` node has as shape the elementwise minimum of the shapes of its arguments. This is analogous to the behaviour of the standard Haskell `zip` function, which produces a list of pairs as long as its shortest argument. Extraneous elements in a larger argument array are simply ignored.
- `Reshape` takes an array and a new shape, and it rearranges the array elements according to the new shape. The new shape must have the same number of elements as the original shape; this is a dynamic check.
- `Replicate` inserts new dimensions into the shape of an array. For example, replicating an array with shape $(((), 2), 3)$ with slice $(((((), 6), ()), 7), ())$ produces an array with shape $(((((), 6), 2), 7), 3)$ where element $(((((), i), j), k), \ell)$ has the value of element $(((), j), \ell)$ of the original.
- `Slice` removes dimensions where `Replicate` adds them: slicing an array with shape $(((((), 6), 2), 7), 3)$ with slice $(((((), ()), 1), 5), ())$ produces an array with shape $(((), 6), 3)$ where element $(((), i), j)$ has the value of element $(((((), i), 1), 5), j)$ of the original.
- `Generate` takes a shape and an expression function assigning a value to each index of the shape. This is the basic "build" constructor of pull-arrays, as described in Section 2.3.1.
- `Foldl` reduces the inner dimension (right-most element in its shape) using the given binary function, thereby reducing the rank of the array by one. The inner dimension is considered extended at index -1 with the initial value. The binary function must be *associative* to allow an efficient parallel implementation. Note that this requirement means that the only

property that makes `Foldl` a *left* fold, as opposed to a right fold that would be named `Foldr`, is that the initial element is added on the left side.

`Foldl` is very much comparable to Haskell's `foldl`, except that the fold reduces just the inner dimension of the array and the combination function must be associative. Idealised Accelerate does not have a fold primitive without this associativity requirement; such a primitive would necessarily be implemented sequentially. Accelerate itself implements a sequential fold using a loop. For a brief discussion about loops, which we unfortunately can not (yet) differentiate, see Section 3.2.1.

- `Sum` is a specialisation of `Foldl` with binary function + and initial value 0, included because it is common and its derivative is significantly simpler.
- `Scanl` is a scan along the inner dimension of the array, enlarging the inner dimension with one element. Element i in one inner slice of the result array is the fold, using the given binary function, over the vector formed by prepending the initial value (the *exp* argument) to elements $0, \ldots, i-1$ of that same inner slice. The binary function must be *associative* to allow an efficient parallel implementation.

  `Scanl` is very much comparable to Haskell's `scanl`, except that the scan traverses just the inner dimension of the array and the combination function must be associative.
- `Scanr` is similar to `Scanl`, except the fold is from the right instead of from the left, and the initial element goes after the end instead of before the beginning.
- `Permute` is a *scatter* operation: procedurally speaking, for `Permute comb def f arr`, we start with `def` and then, for every successive element x at index idx in `arr`, we replace the element y at index (snd (`f` idx)) in `def` with `comb` y x if (fst (`f` idx)) is `true`. The combination function `comb` must be a commutative function to allow an efficient parallel implementation.[33]
- `Backpermute` is a *gather* operation: `Backpermute sz f a` is semantically equivalent to `Generate sz (Lam i (Index a (f i)))`, but it is more restricted and thus easier to analyse.

3.2 Input/output specification of the algorithm

To be able to define an algorithm, one must first know what the algorithm takes as input and must produce as output. For the reverse AD algorithm for array programs, the input is an *afun* value (see Fig. 1) of type $T \to f[()]$, where f is an arbitrary floating-point type and T is an arbitrary array tuple type. This array function is in AST form: it is important to realise that we are given the program as written in the language of Section 3.1, not just a black box. We restrict to a zero-dimensional single-scalar output array because if we do not, the output may contain multiple scalar values, requiring us to vectorise the AD transformation. While the usefulness of such functionality is not disputed, our algorithm handles just the basic case where the output value of the program is a single scalar value.

Let F be the input function (of type $T \to f[()]$). For any reverse AD algorithm[34], including the one presented here, the output is then an *afun* value of type $T \to T$, say dF, with the requirement that for all x of type T for which F is differentiable at x, $dF(x)$ is the gradient of F when evaluated at x. This is to say that for every projection function π of type $T \to \sigma$, where σ is a single-type (i.e. a non-tuple scalar type), we have that $\pi(dF(x)) = \frac{\partial F(x)}{\partial(\pi(x))}$.

[33] In practice, `Permute` is implemented in parallel using atomic updates.

[34] Many reverse AD algorithms return the original function value in addition to the gradient. Because this is easy to add to the complete algorithm, we present it in simplified form where we return only the gradient; our implementation currently makes the same simplification.

The allowance that $dF(x)$ needs to be the gradient function of $F(x)$ only if F is differentiable at x, is necessary (indeed, at other x the gradient is not even defined) and shared with all reverse AD implementations. A crucial consequence of this allowance is that a function that produces a value of discrete type, for example an integer or a tuple of integers, will always have gradient zero with respect to its arguments: its codomain is discrete, so the function is locally constant whenever it is differentiable. Additionally, if all of a function's input arguments are discrete-typed, its domain is discrete and hence the same conclusion applies: its gradient is also always zero. For this reason, primitive functions either from or to discrete types "break" the derivative flow in the program: the expression function `Lam x (PrimApp ToFloating (PrimApp Round (Evar x)))` is allowed to be transformed to the gradient function `Lam x (Const 0.0)`.

3.2.1 Limitations

The algorithm as presented in the following sections can differentiate a large part of the input language, but there are a few elements that are unsupported. These are as follows:

- `Scanl` and `Scanr` cannot currently be differentiated, but will be produced in the derivative of `Foldl`. Fixing this is expected to be a matter of finding the correct derivative of a scan to be inserted in Table 2. If this derivative is found, we do not expect further difficulties in supporting scans for differentiation.
- `Permute` cannot currently be differentiated, but will be produced in the derivative of `Backpermute` and array indexing. While it seems like defining an efficient derivative for `Permute` may be difficult, if it exists and is found, integrating it into the algorithm should not be more difficult than for the scans in the previous point.
- The initial expression and combination function for a `Foldl` cannot contain array indexing operations. This ensures that they cannot contribute an adjoint to any other array value, meaning they can be ignored in the array AD algorithm. Furthermore, we need that the array argument to a `Foldl` has a floating-point type as element type.

  This is a limitation of the derivative we formulate for `Foldl` in Section 3.4.4 (see Table 2). However, we do not expect that it is impossible to give a more general derivative that removes the requirements given here; we were not able to do this due to time constraints.

  Additionally, we do not prove the correctness for the `Foldl` derivative that we do give. However, as shown in Section 4, it has been tested to work in a number of situations.

There are also some features of the full Accelerate language that Idealised Accelerate does not support. These, and other significant differences between the two languages, are listed here.

- Major extension: Accelerate allows unbounded loops on both the array and the expression level using the `awhile` and `while` primitives. Extending the AD algorithm to support loops is expected to be more invasive than only adding the derivative for a new primitive: in order to meet the time complexity requirements for reverse AD, all (or most) of the intermediate values of the computation must be stored for the dual pass. Unbounded iteration means an unbounded number of intermediate values that are incrementally generated, and storing these requires a language primitive that allows for incrementally filling an array, which Accelerate does not yet have. Bounded iteration, where the precise number of iterations is specified before the loop is entered, is easier because the size of the array to allocate is known beforehand, but we still lack a way to incrementally fill an array.
- Minor extensions: Accelerate has some specialised stencil operations for which we have not found a derivative. Since these primitives are not needed to express the other derivatives in this work, we excluded them from the language. The same holds for a fold without initial value, called `foldl1` in the Accelerate surface language.

Accelerate supports foreign function calls that are treated as black-box functionality by the compiler. To be able to differentiate programs containing such calls, the programmer will have to supply an explicit derivative for these operations. Our current algorithm does not support explicit derivatives for subprograms, but adding such support is not expected to be a major change to the algorithm. Indeed, it should be similar to adding a new primitive with a new derivative to the language.

- Additions: While the Accelerate surface language has a `sum` function, its internal representation expresses a sum simply using a `Foldl` with addition as the combination function. Because its derivative is simple and far more efficient than the generic `Foldl` derivative, and furthermore `sum` is common in actual code, we include a specialised `Sum` primitive.
- Program representation: While the Accelerate surface language has tuple projection functions `fst` and `snd`, these are gone in the internal AST. Instead, let-bindings automatically destructure tuples into individual arrays (or in an expression, individual single-typed scalars), meaning that variables in the internal AST are never tuple-typed. While this presented some difficulty when implementing our algorithm in the Accelerate compiler, no real obstacles occurred: adding virtual tuple projection operators at let-bindings allows applying most of the algorithm unchanged. For more details on the implementation, see Section 4.

  The expression functions in Idealised Accelerate always take exactly one argument, possibly tuple-valued; in Accelerate, some expression functions take multiple arguments (e.g. in `Permute` and `ZipWith`). Fortunately, (un)currying gives an equivalence between these two representations.

  In Accelerate, the second expression function argument to `Permute` returns the type `Maybe` s instead of $(\texttt{bool}, s)$. To avoid the question of representing sum-types here, we opted for the plainer alternative. However, we note that because the algorithm can differentiate conditionals, adding support for sum types with case distinction should not be a very invasive change.

Note that Idealised Accelerate does not include a fold without an associativity requirement on the combination function (that is to say, a sequential fold), an operation that is required to implement e.g. recurrent neural networks (RNNs) such as long-short-term memory (LSTM) networks, as well as other loop-like operations. Accelerate itself allows implementing a sequential fold using the looping structures mentioned in the above bulleted list. Supporting loops in the reverse AD algorithm would allow the differentiation of these recurrent networks.

3.3 Example programs

In this section, we will discuss two examples:

1. The first example (Section 3.3.1) is for a simple array program, serving to illustrate the overall form of a derivative of an array program.
2. The second example (Section 3.3.2) is for a more complicated array program that includes array indexing. Having seen how to differentiate such a program in a simple case will be helpful in understanding the general form in the algorithm.

3.3.1 Simple array program

The first array program is shown in Fig. 2. This program is an array function, taking a vector (one-dimensional array) of floating-point values and producing a scalar (a zero-dimensional array) containing one floating-point value. We elide explicit type annotations in the presentation of Idealised Accelerate programs for brevity. A potentially more readable equivalent program in the surface syntax of Accelerate is also provided in the same figure.

Abstract syntax of Idealised Accelerate

```
Alam a
    (Sum₁ (ZipWith₂ (Lam x (PrimApp Mul (Pair (Fst (Evar x)) (Snd (Evar x)))))
                    (Map₃ (Lam x (PrimApp Mul (Pair (Evar x) (Evar x)))) (Avar₅ a))
                    (Map₄ (Lam x (PrimApp Add (Pair (Evar x) (Const 1.0)))) (Avar₆ a))))
```

Concrete Accelerate syntax

```
\a -> sum (zipWith (*) (map (\x -> x * x) a) (map (+1) a))
```

Figure 2: An example program in Idealised Accelerate, and its equivalent in Accelerate. This is an array function of type $f[((), \texttt{int})] \to f[()]$ for some floating-point type f. In Accelerate, that type would be written as follows: `Floating f => Array (Z :. Int) f -> Array Z f`.

The input array is used twice, with both of the `Map` invocations taking it as an argument. One of the `Map`s squares every value, and the other adds 1 to each value. The `ZipWith` then computes an array with $x^2 \cdot (x + 1)$ for all values x in `a`; these values are then summed into the final result.

What is the gradient of this function? Let F be the function in question and let a be an input vector. Write "a indexed at index i" as a_i. Because a is a vector of floating-point values, the gradient of F is the vector of adjoints: at index i should be the value $\frac{\partial F(a)}{\partial a_i}$, which in this case is equal to:

$$\frac{\partial F(a)}{\partial a_i} = \frac{\partial}{\partial x}(x^2 \cdot (x + 1)) = 3x^2 + 2x$$

Hence, we expect the gradient program be semantically equivalent to this expression (though the way it computes it may look different). One way to write a correct gradient program for F is thus to directly implement this expression:

```
-- Shorthand: ADD x y = PrimApp Add (Pair x y)
--            MUL x y = PrimApp Mul (Pair x y)
Alam a (Map (Lam x (ADD (MUL (Const 3.0) (MUL (Evar x) (Evar x)))
                        (MUL (Const 2.0) (Evar x)))))
            (Avar a))
```

However, in the same way the original program was equivalent to a similar simplified program with only a `Sum` and a `Map`. How do we construct a correct gradient program in a somewhat more principled way?

The general theory of reverse AD (see Section 2.1.2) tells us that running the program "backwards" and computing the adjoint of each subexpression will, in the end, produce the adjoint of the input, which is the gradient we seek. First note that while the *adjoint* of a value has a different meaning than the value itself (it is, indeed, the partial derivative of the end result with respect to that value), it does have the same *type*. If x is in $\mathbb{R}$ (e.g. represented in practice using a floating-point value), then $\frac{\partial E}{\partial x}$ for some $E \in \mathbb{R}$ is again a value in $\mathbb{R}$. More generally, if a is an array of values in $\mathbb{R}^n$, then $\frac{\partial E}{\partial a}$ for some $E \in \mathbb{R}$ is again in $\mathbb{R}^n$: each of the individual elements has a derivative. If the value is more complex and contains non-differentiable elements, like integral values, then we give those values adjoint 0 of their own type; doing this allows us to preserve the invariant that the adjoint of a value has the same type as the value itself. This invariant is useful when formulating the full differentiation algorithm in Sections 3.4 and 3.5.

So back to the example, let us apply this "backwards" program execution to our function F. The function body contains 6 array primitives: `Sum`, `ZipWith`, two `Map`s and two `Avars`; in Fig. 2, these have been given the labels 1–6 for easy identification. Let E_i denote the subprogram corresponding to the array primitive labeled i, for example: E_2 is `ZipWith (Lam x ...) (Map ...) (Map ...)`. The return value of F (which is a zero-dimensional array) has, by definition, adjoint 1. Therefore, $\frac{\partial (E_1)_()}{\partial (E_1)_()} = 1$; recall that we use the notation $(-)_i$ to index the array "$-$" at index i. So let us start building the gradient program:

```
Alam a (Alet d1 = Generate Nil (Lam x (Const 1.0))
        in ...)
```

Here `...` indicates that the rest of the program is still to be written. We will use the variables `d`i to store $\frac{\partial(E_1)_()}{\partial(E_i)_j}$ for all relevant indices j, thus we start with the 1 in `d1`.

What we want to know in the end is the adjoint of the input array. Since the input array is used in two different places, this adjoint is a sum of two terms: $\frac{\partial(E_1)_()}{\partial(E_5)_((),i)} + \frac{\partial(E_1)_()}{\partial(E_6)_((),i)}$ for all i from 0 up to the size of a (minus 1). But to compute the adjoints of E_5 and E_6, we first have to compute the adjoints higher up in the tree, starting with E_2, the argument to the `Sum`.

Since $\frac{\partial}{\partial x}(x+y) = \frac{\partial}{\partial y}(x+y) = 1$, and more generally $\frac{\partial}{\partial x_i}(x_1 + \cdots + x_n) = 1$, the adjoints of the elements of the input array to a `Sum` are the same as those of the corresponding elements of its own adjoint; in this case: $\frac{\partial(E_1)_()}{\partial(E_2)_i} = \frac{\partial(E_1)_()}{\partial(E_1)_()} = \texttt{d1}_()$. Hence, the program continues like this:

```
Alam a (Alet d1 = Generate Nil (Lam i (Const 1.0))
        in Alet d2 = Generate (Shape a) (Lam i (Index d1 (Fst (Evar i))))
        in ...)
```

Note that `Fst` on an index value strips off the last component; in this case, `Fst (Evar i)` becomes just `Nil`.

Having computed the adjoint of the `ZipWith`, we use `ZipWith`'s derivative to compute the adjoints of its arguments. Since $(\texttt{ZipWith}\ f\ a\ b)_i = f(a_i, b_i)$, we get $\frac{\partial(E_1)_()}{\partial(E_3)_i} = \frac{\partial(E_1)_()}{\partial(E_2)_i} \cdot \frac{\partial(E_2)_i}{\partial(E_3)_i} = \texttt{d2}_i \cdot \frac{\partial}{\partial x} f(x,y)$, and analogously $\frac{\partial(E_1)_()}{\partial(E_4)_i} = \texttt{d2}_i \cdot \frac{\partial}{\partial y} f(x,y)$. Here we have $f(x,y) = x \cdot y$, so we get $\frac{\partial}{\partial x} f(x,y) = y$ and $\frac{\partial}{\partial y} f(x,y) = x$. Note that these refer to the original function inputs; this is to be expected as explained in Section 2.1.2, since we need some of the intermediate values from normally computing F when computing its gradient. In other words: we need some of the primal results when computing the dual.

If we prefix the necessary part of the primal computation before our gradient program, we can then append the adjoint of the arguments of the `ZipWith`:

```
Alam a (-- First compute (a part of) the primal computation
        Alet t5 = Avar a
        in Alet t3 = Map (Lam x (PrimApp Mul (Pair (Evar x) (Evar x)))) (Avar t5)
        in Alet t6 = Avar a
        in Alet t4 = Map (Lam x (PrimApp Add (Pair (Evar x) (Const 1.0)))) (Avar t6)
        -- Then compute the adjoints in the dual computation
        in Alet d1 = Generate Nil (Lam i (Const 1.0))
        in Alet d2 = Generate (Shape a) (Lam i (Index d1 (Fst (Evar i))))
        in Alet d3 = ZipWith (Lam pair (PrimApp Mul (Pair
                                        (Fst pair)     -- adjoint from d2
                                        (Snd pair))))  -- y value from t4
                             (Avar d2) (Avar t4)
        in Alet d4 = ZipWith (Lam pair (PrimApp Mul (Pair
                                        (Fst pair)     -- adjoint from d2
                                        (Snd pair))))  -- x value from t3
                             (Avar d2) (Avar t3)
        in ...)
```

Notice how we pass only the "other" primal argument to the `ZipWith`s introduced for the adjoints of the argument to node 2 of F. This works here because the partial derivatives of the expression function in node 2 only refer to some of its original inputs, but in general both `t3` and `t4` would need to be passed (zipped). Furthermore, the lambdas written here in the gradient program can be simplified to `Lam pair (PrimApp Mul pair)`, but to illustrate the more general case, this simplification is not made here.

Now we have the adjoints of the `Map`s, nodes 3 and 4, what remains is those of nodes 5 and 6 and the input. The `Map` expression functions, $f_1(x) = x^2$ and $f_2(x) = x + 1$, have derivatives $\frac{\partial f_1(x)}{\partial x} = 2x$ and $\frac{\partial f_2(x)}{\partial x} = 1$, so we get $\frac{\partial (E_1)_{()}}{\partial (E_5)_i} = \texttt{d3}_i \cdot 2\texttt{t5}_i$ and $\frac{\partial (E_1)_{()}}{\partial (E_6)_i} = \texttt{d4}_i$, where `t5` is the primal value of node 5. This appends the following to our growing program:

```
    ...
    in Alet d5 = ZipWith (Lam pair (PrimApp Mul (Pair
                                        (Fst pair)  -- adjoint from d3
                                        (PrimApp Mul (Pair (Const 2.0) (Snd pair))))))
                         (Avar d3) (Avar t5)
    in Alet d6 = Avar d4
    in ...
```

Now that we know the adjoints of node 5 and node 6, the variable references of the input, the adjoint of the input is the elementwise sum of `d5` and `d6`, as argued earlier. This means that the final program is as follows:

```
-- Shorthand: ADD x y = PrimApp Add (Pair x y)
--            MUL x y = PrimApp Mul (Pair x y)
Alam a (Alet t5 = Avar a
        in Alet t3 = Map (Lam x (MUL (Evar x) (Evar x))) (Avar t5)
        in Alet t6 = Avar a
        in Alet t4 = Map (Lam x (ADD (Evar x) (Const 1.0))) (Avar t6)
        in Alet d1 = Generate Nil (Lam i (Const 1.0))
        in Alet d2 = Generate (Shape a) (Lam i (Index d1 (Fst (Evar i))))
        in Alet d3 = ZipWith (Lam pair (MUL (Fst pair) (Snd pair))) (Avar d2) (Avar t4)
        in Alet d4 = ZipWith (Lam pair (MUL (Fst pair) (Snd pair))) (Avar d2) (Avar t3)
        in Alet d5 = ZipWith (Lam pair (MUL (Fst pair) (MUL (Const 2.0) (Snd pair))))
                             (Avar d3) (Avar t5)
        in Alet d6 = Avar d4
        in ZipWith (Lam pair (PrimApp Add pair)) (Avar d5) (Avar d6))
```

This array function computes the gradient of the array function in Fig. 2. The method of labeling the AST nodes and computing their adjoints in turn is the same as what we will apply in the full algorithm.

Efficiency. The gradient program produced here contains many array primitives and looks a lot less efficient than the "naive" gradient program we derived earlier in this section. However, we are saved by array fusion: for example, the `Index d1 (Fst (Evar i))` in `d2` simplifies to `Index d1 Nil`, which fuses to `Const 1.0`. This can then be inlined into the rest of the program, and the same kind of trick can be applied to, for example, a `Map` argument to a `ZipWith`. The Accelerate compiler implements this kind of array fusion optimisation [37], and can fuse this whole program down to:[35]

```
\a -> generate (shape a) (\i -> let x = a ! i in 2.0 * x + 3.0 * x * x)
```

3.3.2 Array program with indexing

In the previous section we studied how to compute a gradient function for a simple array function: label all the array nodes in the AST with unique labels and compute the primal value of all (necessary) nodes, and then compute the adjoint for each node. If the array function in question contains an expression that performs array indexing, the overall method still applies, but the array node containing that expression will make some extra adjoint contributions beyond those to its arguments.

[35] In practice, Accelerate currently does not quite get the program to this form because of an unresolved issue with the fusion implementation. This is expected to be resolved with the upcoming complete redesign of this implementation.

Abstract syntax of Idealised Accelerate

```
Alam a
    (Sum1 (Map2 (Lam x (PrimApp Add (Pair
                                (Evar x)
                                (Index a3 (Pair Nil (PrimApp Round (Evar x))))))
                (Avar4 a)))
```

Concrete Accelerate syntax

```
\a -> sum (map (\x -> x + a ! I1 (round x)) a)
```

Figure 3: An example program in Idealised Accelerate, and its equivalent in Accelerate, of type $f[((), \texttt{int})] \to f[()]$ for some floating-point type f.

Consider the array program in Fig. 3, which takes a vector a of floating-point values and produces a single floating-point value.[36] For each element x in a, it adds x and the element of a at the index round(x), and finally returns the sum of all those addition results. The program assumes that all those indexing operations are in bounds; checking or ensuring this is beyond the scope of differentiation, but we must ensure that for any input for which the original program does not perform out-of-bounds array indexes, the gradient program we produce also does not.

Note that in Fig. 3, the reference to the array variable `a` inside the expression also has a label, in addition to the actual array nodes in the AST. This is because that variable reference will actually be collecting an adjoint.

Let us start the gradient program as usual, first computing some of the primal values and then computing the adjoints of nodes 1, 2 and 4:[37]

```
Alam a (Alet t3 = Avar a
        in Alet t4 = Avar a
        in Alet t2 = Map (Lam x (PrimApp Add (Pair
                                    (Evar x)
                                    (Index t3 (Pair Nil (PrimApp Round (Evar x)))))))
                         (Avar t4)
        in Alet d1 = Generate Nil (Lam i (Const 1.0))
        in Alet d2 = Generate (Shape t2) (Lam i (Index d1 (Fst (Evar i))))
        in Alet d4 = ZipWith (Lam pair (Fst pair))  -- return input adjoint
                             (Avar d2) (Avar t4)
        in Alet d3 = ...
        in ...)
```

Note that we visit the nodes in the order 1, 2, 4, 3; an alternative valid order would have been 1, 2, 3, 4. Both are valid topological orderings of the dependency graph that is induced by the program.

The gradient of the expression function in the `Map` is 1 (or equivalently, its derivative function is the identity function) because the only subexpression contributing a non-zero adjoint to `x` is the left-hand side of the `Add` operation. The `Round` primitive returns an integral result and thus does not contribute anything to its argument `x`. For this reason, the adjoint contribution to node 4 (i.e. `d4`) is the same as `d2`; for generality, however, we write the usual `ZipWith` taking the incoming adjoint and the primal argument.

[36] The concrete Accelerate syntax in Fig. 3 uses one construct of which the semantics may not be evident, namely the `I1` syntax. This is a bidirectional pattern synonym [44]. In an expression context (which is how it occurs here), `I`n constructs the index type of an n-dimensional array given n arguments; this means that `I1 i` corresponds to `Pair Nil i` in Idealised Accelerate. For more information on these pattern synonyms and specifics of Accelerate's surface language, see the Accelerate documentation [12] or the relevant publication [38].

[37] Here and in the previous section, some values from the primal computation that happen to not be necessary in the dual computation are elided from the gradient program. The full algorithm defined later will always compute all primal nodes, but this can be reduced to the form of these examples using dead-code elimination.

What is the `Map`'s adjoint contribution to node 3? Write `t3` and `t4` for the primal results of nodes 3 and 4, respectively. (They happen to be the same here, because they are both a variable reference to `a`; however, the following text does not use that information.) For every element x in `t4`, the expression function will be executed, which performs an array indexing operation on `t3` at index round(x). The result of each runtime invocation of this array indexing node, of which there is exactly one per invocation of the surrounding expression function,[38] has itself an adjoint that expression AD (analogous to the array AD discussed up until now) wants to propagate to the value in the array being indexed (namely, `t3`). We will collect every one of these individual contributions in an array the size of `t3`, which will be the adjoint of node 3 (i.e. `d3`); to do that we will use the `Permute` array primitive.

Recall from Section 3.1 that `Permute` takes four arguments:

1. `comb`: The (commutative) combination function used to combine a value being added to a cell with the value already in that cell;
2. `def`: The destination array, filled with default values;
3. `f`: An expression function that maps an index in the source array `src` to an index in the destination array `def`, together with a boolean that when `false` prevents this source element from being mapped;
4. `src`: The source array of values to be mapped.

In this case, the combination function is (+) and the array of default values is an array the size of `t3` filled with zeros, since we want every cell of the result to contain the *sum* of the contributions produced for that cell. The mapping function `f` maps an index `i` in `t4` to `Pair Nil (PrimApp Round (Index t4 i))`, which is the target index of the array indexing operation in the expression function invocation at index `i` (recall the code in Fig. 3). The source array should contain the adjoints of the indexing operation in the expression function, which here happen to be equal to those found in `d2` because of the simplicity of the expression function.

This means we can compute `d3` and complete the program in the following way:

```
1  Alam a (Alet t3 = Avar a
2          in Alet t4 = Avar a
3          in Alet t2 = Map (...) (Avar t4)  -- elided for brevity (same as before)
4          in Alet d1 = Generate Nil (Lam i (Const 1.0))
5          in Alet d2 = Generate (Shape t2) (Lam i (Index d1 (Fst (Evar i))))
6          in Alet d4 = ZipWith (Lam pair (Fst pair)) (Avar d2) (Avar t4)
7          in Alet d3 = Permute (Lam pair (PrimApp Add pair))
8                               (Generate (Shape t3) (Lam i (Const 0.0)))
9                               (Lam i (Pair (Const true) (Pair Nil (PrimApp Round (Index t4 i)))))
10                              (Avar d2)
11         in ZipWith (Lam pair (PrimApp Add pair)) (Avar d3) (Avar d4))
```

However, we could only write this `Permute` like this because we could easily recompute the index into `t4` as $((), \text{round}(x))$ and because we knew that the adjoint of an invocation of the `Index` node in the program happens to be the same as that of the surrounding expression function. In general, however, we want to assume neither of these things. To not make these assumptions, we want the expression lambda that computes the adjoint of the argument of node 2 (the `Lam pair (Fst pair)` on line 6) to also return the target index of the indexing operation, as well as the contribution that should be added there. Thus we change line 6 of the above program to the following:

[38] This can stop being true in the presence of conditionals (`Cond`), in which case the number of invocations can be zero as well as one.

```
-- Array of: (adjoint of argument of node 2, (adjoint of index operation, target index))
in Alet res2 = ZipWith (Lam pair
                           (Pair (Fst pair)
                                 (Pair (Fst pair)
                                       (Pair Nil (PrimApp Round (Snd pair))))))
                       (Avar d2) (Avar t4)
in Alet d4 = Map (Lam p (Fst p)) (Avar res2)
in Alet idx2 = Map (Lam p (Snd p)) (Avar res2)  -- Array of: (adjoint of index op., target index)
```

Note that in this case the adjoint of the argument (i.e. the adjoint for node 4) and the adjoint of the index operation are the same (both `Fst pair` in the code), but this need not be so, and the rest of the code is oblivious of this coincidence.

Indeed, now all the information that we need to compute the adjoint of node 3 is in the variable `idx2`. Thus we can change the `Permute` in the computation of `d3` (lines 7–10 in the previous code fragment) to use `idx2`, and arrive at the following complete program:

```
1  Alam a (Alet t3 = Avar a
2          in Alet t4 = Avar a
3          in Alet t2 = Map (Lam x (PrimApp Add (Pair
4                                       (Evar x)
5                                       (Index t3 (Pair Nil (PrimApp Round (Evar x)))))))
6                           (Avar t4)
7          in Alet d1 = Generate Nil (Lam i (Const 1.0))
8          in Alet d2 = Generate (Shape t2) (Lam i (Index d1 (Fst (Evar i))))
9          in Alet res2 = ZipWith (Lam pair
10                                     (Pair (Fst pair)
11                                           (Pair (Fst pair)
12                                                 (Pair Nil (PrimApp Round (Snd pair))))))
13                                 (Avar d2) (Avar t4)
14         in Alet d4 = Map (Lam p (Fst p)) (Avar res2)
15         in Alet idx2 = Map (Lam p (Snd p)) (Avar res2)
16         in Alet d3 =
17             Permute (Lam pair (PrimApp Add pair))
18                     (Generate (Shape t3) (Lam i (Const 0.0)))
19                     (Lam i (Pair (Const true)
20                                  (Snd (Index idx2 i))))  -- Where to map to? The target index.
21                     (Map (Lam p (Fst p)) (Avar idx2))  -- What to map? The index operation's adjoint.
22         in ZipWith (Lam pair (PrimApp Add pair)) (Avar d3) (Avar d4))
```

This computes the same result as the previous full gradient program, but illustrates how we can make the transformation more systematic while avoiding recomputation.

Observations and reflections. When constructing the gradient programs in the previous two examples, the process we applied consisted of three traversals of the array AST: one to label the nodes, one to build the "let-stack" that computes the primal values, and one to build the let-stack that computes the adjoints. Within the traversals, we applied only *local reasoning*:

- A primal value can be computed using just the information in that node and the knowledge that its array arguments have already been computed and are available using variables that are in scope.
- For the adjoints: if, throughout the traversal, we keep a map from array node labels to the list of contributions that have been produced for them, the adjoint of a node N can be computed just by summing the terms in the list at key N in that contribution map. The only further task is then to update the map at each node, which just involves looking at the node's arguments and computing the correct partial derivative with respect to those arguments. The result should then be added to the contribution map for use when visiting the argument nodes, or in the case of a variable reference, the right-hand side that it refers to.

Even though the code that results from the program transformation has control flow that can be seen as going in the reverse direction of the original program flow, the *construction* of that program is very structure-oriented and follows the shape of the original program's AST.

An earlier version of the algorithm first converted the program to a graph form where node arguments are incoming edges and node usages are outgoing edges. It then generated the primal and dual let-stacks using traversals over that graph instead of the original AST. This is a significantly more invasive pass over the program, changing the representation of "value reuse" in the program from multiple occurrences of the same variable reference to multiple incoming edges of a graph node. While this was initially felt to be a more intuitive representation to define a reverse AD algorithm on, in the end the intermediate graph representation turned out to be unnecessary and in fact counterproductive for supporting the use of conditionals in the input program. We hope that the algorithm presented in this document is simpler and more pragmatic.

3.4 Array transformation

The array AD algorithm consists of a number of steps, that have each already been illustrated in the examples in Sections 3.3.1 and 3.3.2.

- First the array nodes in the program's AST are *labeled* in order to be able to refer to them. These labels were indicated as subscripts in the input programs, for example in Fig. 3. The labeling specifics will be discussed in Section 3.4.2.
- Then, the primal values are computed in the *primal phase*, discussed in Section 3.4.3. These are the `t`N variables in the code snippets in the examples.
- The dual values are then computed in the *dual phase*, discussed in Section 3.4.4. These are the `d`N variables in the examples, containing the intermediate adjoints.
- Finally, we collect the gradient of the input value at the bottom of the let-stack and put the whole gradient program together; this we do in Section 3.4.5.

A core component of both the primal and the dual phase is how array AD uses the expression AD algorithm to handle the differentiation of expression functions that appear in the program. For this reason we first describe the interface to expression AD that is used in Section 3.4.1, after which we will continue with the main description of the array AD algorithm.

3.4.1 Interface of expression AD

Because each array AST node is visited twice, once in the primal and once in the dual pass, the primal and dual transformations of an expression function need to be split from each other, forming two halves. Furthermore, handling array indexing (and producing the correct `Permute` array primitives) requires that the expression AD algorithm tells us about any `Index` operations happening in the expression function's body; this is similar to how we generalised the handling of array indexing at the end of Section 3.3.2. For these reasons, a simple interface for expression AD that just returns the gradient-producing function is insufficient.

Instead, array AD will use the interface to expression AD that we define here. This interface uses two types t_{tmp} and t_{idxadj} that are built up inside the expression AD algorithm; their meaning will be explained below. Given an expression function F of type $t \to t'$, expression AD returns the following things:

- an expression function of type $t \to (t', t_{\text{tmp}})$: the primal lambda,
- an expression function of type $(t', t_{\text{tmp}}) \to (t, t_{\text{idxadj}})$: the dual lambda. We also get information about the layout of t_{idxadj}; see the description below.

Let us discuss this interface piece by piece. The input is an expression function F of type $t \to t'$; this means that the expression AD interface only works for expression *functions*, not individual expressions. However, this not an insurmountable problem: indeed, one can wrap an expression of type t' into an expression function of type $() \to t'$ and apply the expression AD algorithm as-is.[39] The expression function should be provided, of course, in AST form as defined in Section 3.1, and not as a black box.

The first part of the output of expression AD is the primal lambda: this is an expression function of type $t \to (t', t_{\text{tmp}})$ that given an argument x of type t, returns $F(x)$ as well as a tuple encoding the intermediate values ("temporaries") computed while running F. Although this temporaries tuple did not appear in Section 3.3 to simplify the presentation there, the concept is quite natural: as we saw in the array AD examples, the dual pass needs some values from the primal pass in order to compute the gradient. This structure is the same in expression AD, and the temporaries tuple is the channel via which the primal lambda communicates those values to the dual lambda. While we could have chosen to let the dual lambda from expression AD recompute the required primal values, in this exposition we instead opt for preventing as much recomputation as possible.[40]

The second part of the output is the dual lambda of type $(t', t_{\text{tmp}}) \to (t, t_{\text{idxadj}})$. This is an expression function taking the adjoint of the return value of F for some invocation of F, as well as the temporaries tuple produced when running the primal lambda on the corresponding input value. Note that the type of the adjoint of the return value of F is the same as the type of the return value itself; this is a special case of the general rule that the adjoint of a subexpression has the same type as that subexpression itself (see Section 3.3.1).

The output of the dual lambda is a pair containing the adjoint of its input value as well as the *index adjoint information tuple* (IAI tuple) of type t_{idxadj}. This tuple contains, for every occurrence of an `Index` node in F, three values: its target index in the indexed array as computed in the primal pass, the adjoint of that `Index` node as computed in the dual pass, and a boolean indicating whether the `Index` node was executed at all. (An `Index` node may remain unexecuted if it is inside an untaken `Cond` branch.) The boolean is also computed in the primal pass: the choice of whether to execute the `Index` happens in the primal pass, and this information is passed on to the dual pass via the temporaries tuple, from where it is passed on to array AD via the IAI tuple.

Since array AD will need to use the values in this IAI tuple to create the correct `Permute` operations, the output of expression AD additionally describes *which* arrays are indexed in F, and *where* in the index adjoint information tuple the entries for that array can be found. The exact representation of this description is an implementation detail that we will abstract over in this discussion.

Example. Let us look at an example to illustrate the expression AD interface before we continue. Consider the following expression function of type $(\texttt{float}, \texttt{int}) \to \texttt{float}$ that multiplies its first argument with the value in the array `a` at the index specified by the second argument, where `a` has type $\texttt{float}[((), \texttt{int})]$:

```
Lam pair (PrimApp₁ Mul (Pair (Fst₂ (Evar pair)) (Index₃ a (Pair₄ Nil (Snd (Evar pair))))))
```

In concrete Accelerate syntax, this expression function can be written as the more readable `\p -> fst p * a ! I1 (snd p)`. Note that some of the expression nodes have been labeled for

[39] The gradient result from differentiating an expression is not very useful (the gradient of () is ()), but one may want to use the index adjoint information in t_{idxadj}.

[40] In a parallel setting like on a GPU device, it is not clear a priori that it is beneficial to store intermediate values of small expression functions to prevent a tiny amount of recomputation: by doing so we exchange some processing time for (potentially quite expensive) additional memory usage and traffic. While we will briefly return to this topic in Section 5, this is mostly reserved for future work.

easy reference; more would be labeled by the actual expression AD algorithm, which will be explained in Section 3.5.

The primal lambda might then look as follows (the full algorithm produces a somewhat more verbose program):

```
Lam pair
  (Let t2 = Fst (Evar pair)                -- type: float
   in Let t4 = Pair Nil (Snd (Evar pair))  -- type: ((), int)
   in Let t3 = Index a (Evar t4)           -- type: float
   in Let t3exec = Const true              -- type: bool
   in Let t1 = PrimApp Mul (Pair           -- type: float
                   (Evar t2) (Evar t3))
   in Pair t1
        (Pair (Pair (Pair (Pair (Pair Nil (Evar t1)) (Evar t2)) (Evar t3)) (Evar t3exec)) (Evar t4)))
```

This function has type $(\mathtt{F}, \mathtt{I}) \to (\mathtt{F}, ((((((), \mathtt{F}), \mathtt{F}), \mathtt{F}), \mathtt{B}), ((), \mathtt{I})))$, writing `F` for `float`, `I` for `int` and `B` for `bool` for brevity. Thus, here we have $t_{\text{tmp}} = ((((((), \mathtt{F}), \mathtt{F}), \mathtt{F}), \mathtt{B}), ((), \mathtt{I}))$. Note that `t3exec` is a boolean indicating whether the `Index` was executed; it is always true in this case.

The primal lambda returns the function result as the first `float` value, and a tuple containing the temporaries as the second element of the output pair. The list of temporaries that needs to be stored in the temporaries tuple is encoded as a nested sequence of pairs, terminated with an empty tuple. This is not the only possible encoding, but it happens to be the one that the Accelerate compiler uses internally when translating longer tuples from the user program into its internal language, which knows only pairs. It also happens to be an encoding that is easy to generate and consume recursively.

The dual lambda for the example function might look as follows:

```
Lam input
  (Let t2 = Snd (Fst (Fst (Fst (Snd (Evar input)))))
   in Let t3 = Snd (Fst (Fst (Snd (Evar input))))
   in Let t3exec = Snd (Fst (Snd (Evar input)))
   in Let t4 = Snd (Snd (Evar input))
   in Let d1 = Fst (Evar input)
   in Let d2 = PrimApp Mul (Pair (Evar d1) (Evar t3))
   in Let d3 = PrimApp Mul (Pair (Evar d1) (Evar t2))
   in Pair (Pair (Evar d2) (Const 0))
           (Pair Nil (Pair (Pair (Evar d3) (Evar t4)) (Evar t3exec))))
```

This function has type $(\mathtt{F}, ((((((), \mathtt{F}), \mathtt{F}), \mathtt{F}), \mathtt{B}), ((), \mathtt{I}))) \to ((\mathtt{F}, \mathtt{I}), ((), ((\mathtt{F}, ((), \mathtt{I})), \mathtt{B})))$; note the re-occurrence of t_{tmp}. Recall that $(\mathtt{F}, \mathtt{I})$ was the input type of the original function, and thus also the gradient type. t_{idxadj} here takes the value $((), ((\mathtt{F}, ((), \mathtt{I})), \mathtt{B}))$, one possible tuple encoding of a one-element list containing a triple ((adjoint, target index), was-executed) of the single `Index` in the original function.

3.4.2 Labeling

The first step in the array AD algorithm is to label the array nodes in the AST with unique labels. The input to the labeling step is an array function, of production *afun* in Fig. 1. In this array function, all nodes of the *arr* production except `Alet` shall get a unique integer label used to refer to them in the primal and dual phases of the algorithm. For examples, see Section 3.3 above.

As stated, the only node type that does not get a label is `Alet`, since it does not compute anything. However, its right-hand side program as well as the body are labeled as usual.

In the algorithm below, we will regularly assume that any given subprogram has a label assigned; while this is not true if the subprogram is an `Alet`, we will then consider the subprogram to have the label of the `Alet`'s body program.

In Section 3.3.2, the variable reference in an `Index` operation in an expression also had a label. In the algorithm we will not do this: only actual *arr* nodes get labels in the array AD algorithm.[41] Instead, the contribution made to the indexed array by the `Index` node will be collected in the contribution map, as we will see below.

For an example, see the following program (expression function bodies elided for conciseness):

```
1  Map₁ (Lam x ...) (Alet a = Use₂ [1.0, 2.0, 3.0] in Map₃ (Lam x ...) (Avar₄ a))
```

If we require the label of the array argument of the `Map` with label 1, the answer is 3: the label of the `Alet` is considered to be the label of its body.

3.4.3 Primal

In the primal phase of the array AD algorithm, we perform a recursive traversal over the AST of the input array program, which is the body of the leading `Alam` in the input. For each visited node, the algorithm will produce a number of *incomplete array-let-bindings* that will be concatenated in the end to produce the result of the primal phase. First we will briefly discuss these incomplete let-bindings, and their generalisation of a *builder*; afterwards, we will describe the primal recursive traversal in detail.

Builders. An *incomplete array-let-binding* is of the form `Alam var = prog in _` for some `var` of production *avar* and some `prog` of production *arr* (Fig. 1). An incomplete array-let-binding can be used to create a valid array program, i.e. a valid AST of production *arr*, by substituting a valid program for the `_`; the intuitive meaning of an incomplete let-binding is thus as a *prefix* to another program that brings a particular computation in scope under a given name.

Analogous to incomplete array-let-bindings we can also speak about incomplete expression-let-bindings, replacing `Alam` with `Lam`, etc. Both forms of incomplete let-bindings are a special case of a *builder*, which is an AST fragment of any production with exactly one hole. Given two builders b_1 and b_2, we can substitute b_2 in the hole in b_1 to obtain a new builder $b_1 \circ b_2$. This composition notation is inspired by the likeness of builders to functions taking an AST and returning a new AST that wraps the input in some larger program: if f_1 is the AST function corresponding to a builder b_1 and likewise for f_2 and b_2, then the function composition $f_1 \circ f_2$ corresponds to the builder $b_1 \circ b_2$.

Of course, it only makes sense to create the composition $b_1 \circ b_2$ if the production required in the hole of b_1 is the same production as the one that b_2 builds. For array AD, we will only see builders for production *arr* (in fact, only compositions of incomplete array-let-bindings); for expression AD in Section 3.5, we will see builders for production *exp*.

As an example, for incomplete let-bindings, their builder composition results in nested let-bindings. If the builder b_1 is `Alet a1 = prog1 in _` and b_2 is `Alet a2 = prog2 in _`, then their composition $b_1 \circ b_2$ is the builder `Alet a1 = prog1 in Alet a2 = prog2 in _`. We will also refer informally to this program shape as a *let-stack*.

Recursive traversal. The primal transformation is a recursive algorithm that takes as input an AST of an array program (i.e. of production *arr* in Fig. 1) and a variable environment (a mapping from array variables in scope to the label of their bound right-hand side), and produces two results: a builder for production *arr* and a set of array variable names, called the *to-store* set. The output of the primal phase is the builder; in the dual phase (Section 3.4.4) we will construct the AST that goes in the hole in this primal builder. The to-store set is only used internally in the primal phase: we use it to handle conditionals correctly.

[41] Expression AD, of course, will label *exp* nodes instead.

The initial variable environment at the top level contains only a mapping from the `Alam` argument to a special value indicating that this is the *global function argument*.

For all array node types except `Alet` and `Acond`, the primal transformation then follows a recipe described here. First the transformation is recursively invoked on all array subprograms, meaning the *arr* arguments to the current node; the variable environment (the second parameter of the primal phase) is passed unchanged. Each of these recursive calls produces a builder and a to-store set. Furthermore, we rename any array variables occurring in *exp* and *fun* arguments of the current node: if such a variable is mapped to a label n in the environment, it is replaced with `a`n; if it is the global function argument, it is left as-is. We assume this does not create name collisions; if necessary, the global function argument should be renamed to a non-colliding name like "`argument`".

Then we look in the row for the current node in Table 1 to find the corresponding builder and to-store set; the current node is listed in the first column titled "*arr* primitive". The label of the current node is indicated with a subscript n. The m variables should contain the labels of the array arguments.[42] For `Avar`, if the corresponding binding in the variable environment indicates that this is the global function argument, the $\texttt{Avar}_n$ `argument` row is used; otherwise the environment yields the label of the referenced right-hand side, which becomes the value of m in the $\texttt{Avar}_n$ $\texttt{var}_{\to m}$ row.

The second and third column of the table give the builder and to-store set produced for this node. In the second column, the notation $\mathcal{P}(\texttt{fun})$ is used to denote the primal lambda for `fun`, and the notations λ`fst` and λ`snd` are shorthand for `Lam x (Fst (Evar x))` and `Lam x (Snd (Evar x))`, respectively. Composing the builders for the arguments and the builder from the table gives the output builder for this node. The order of composition of the argument builders does not matter, but the builder from the table needs to come last (otherwise the `a`m variable references would be out of scope). Finally, the union of the to-store sets of the arguments together with the one from the table gives the output to-store set of this node. The sets in this union will always be disjoint.

With this, the full primal traversal is specified if the input program uses none of the following: `Alet`, `Acond`, `Scanl`, `Scanr`, `Permute`. As previously stated in Section 3.2.1, for the scans and `Permute` we have not defined a derivative; this means their dual transform is not defined, so we also omit their primal transform here. The behaviour for the `Alet` and `Acond` nodes is described below.

Let-bindings. When visiting an `Alet a =` $\texttt{rhs}_{m_1}$ `in` $\texttt{body}_{m_2}$ node, `rhs` and `body` are recursively traversed producing builders b_{rhs} and b_{body} and to-store sets s_{rhs} and s_{body}. The recursive call for `rhs` gets passed the unchanged environment, while the call for `body` has the mapping `a` $\Rightarrow m_1$ added to the environment. The output builder is $b_{\text{rhs}} \circ b_{\text{body}}$, and the output to-store set is $s_{\text{rhs}} \cup s_{\text{body}}$.

Conditionals. For $\texttt{Acond}_n$ `exp` $\texttt{arr1}_{m_1}$ $\texttt{arr2}_{m_2}$, we cannot use the recipe from above: the crucial difference between `Acond` and the other array node types is that `Acond` introduces *conditional execution*. Therefore, we cannot just flatten execution into a stack of let-bindings, automatically bringing all subexpressions in scope, as we did in the other cases. Instead, informally, we will expand `arr1` and `arr2` inside the branches of an `Acond`, and use the to-store sets to determine what to "export" outside the branches to the main sequence of let-bindings.

The to-store sets are constructed in the rest of the primal algorithm to contain all variables that we might need in the dual phase; hence they also describe what variables we need to retain after closing a conditional branch. In practice, some of the variables we include in to-store sets

[42] Recall that the label of an `Alet` is the label of its body, so all (sub)programs have a well-defined label.

***arr* primitive**	**Builder**	**To store**
$\texttt{Use}_n$ lit	Alet an = Use lit in _	{an}
$\texttt{Avar}_n$ argument	Alet an = Avar argument in _	{an}
$\texttt{Avar}_n$ $\texttt{var}_{\to m}$	Alet an = Avar am in _	{an}
Alet a = rhs in body	(special, see text)	
$\texttt{Anil}_n$	Alet an = Anil in _	{an}
$\texttt{Apair}_n$ $\texttt{arr1}_{m_1}$ $\texttt{arr2}_{m_2}$	Alet an = Apair (Avar am_1) (Avar am_2) in _	{an}
$\texttt{Afst}_n$ $\texttt{arr}_m$	Alet an = Afst (Avar am) in _	{an}
$\texttt{Asnd}_n$ $\texttt{arr}_m$	Alet an = Asnd (Avar am) in _	{an}
$\texttt{Reshape}_n$ exp $\texttt{arr}_m$	Alet an = Reshape exp (Avar am) in _	{an}
$\texttt{Replicate}_n$ exp $\texttt{arr}_m$	Alet an = Replicate exp (Avar am) in _	{an}
$\texttt{Slice}_n$ exp $\texttt{arr}_m$	Alet an = Slice exp (Avar am) in _	{an}
$\texttt{Generate}_n$ exp fun	Alet a$n$res = Generate exp $\mathcal{P}$(fun) in Alet an = Map λfst (Avar a$n$res) in Alet a$n$tmp = Map λsnd (Avar a$n$res) in _	{an, a$n$tmp}
$\texttt{Map}_n$ fun $\texttt{arr}_m$	Alet a$n$res = Map $\mathcal{P}$(fun) (Avar am) in Alet an = Map λfst (Avar a$n$res) in Alet a$n$tmp = Map λsnd (Avar a$n$res) in _	{an, a$n$tmp}
$\texttt{ZipWith}_n$ fun $\texttt{arr}_{m_1}$ $\texttt{arr}_{m_2}$	Alet a$n$res = ZipWith $\mathcal{P}$(fun) (Avar am_1) (Avar am_2) in Alet an = Map λfst (Avar a$n$res) in Alet a$n$tmp = Map λsnd (Avar a$n$res) in _	{an, a$n$tmp}
$\texttt{Sum}_n$ $\texttt{arr}_m$	Alet an = Sum (Avar am) in _	{an}
$\texttt{Foldl}_n$ fun exp $\texttt{arr}_m$	Alet an = Foldl fun exp (Avar am) in _	{an}
$\texttt{Backpermute}_n$ exp fun $\texttt{arr}_m$	Alet an = Backpermute exp fun (Avar am) in _	{an}
$\texttt{Acond}_n$ exp $\texttt{arr1}_{m_1}$ $\texttt{arr2}_{m_2}$	(very special, see text)	
Scanl/Scanr/Permute	(not supported, see text)	

Table 1: Rules for the primal phase of array AD. A subscript m on an argument in the first column means that the label of that subprogram is m; $\texttt{var}_{\to m}$ means that the variable *refers* to a right-hand side with label m. The $\texttt{Avar}_n$ argument case is only used if the variable refers to the global function argument. "λfst" is shorthand for Lam x (Fst (Evar x)), and "λsnd" analogous for Snd. $\mathcal{P}$(fun) means the primal lambda for fun. The column "To store" lists the variable names to store for Acond; see the text.

might actually be unnecessary in the dual phase; we do not optimise this in this work, but some potential options are discussed in Section 6.1.

To describe the algorithm for conditionals, we first have to make some definitions. First replace all array variables in `exp` that are not the global function argument with a reference to am, where m is the corresponding label found in the environment; the result is a renamed expression `expR`. Then recursively transform `arr1` and `arr2`, producing respectively builders b_1 and b_2 and to-store sets s_1 and s_2. For $k \in \{1, 2\}$:

- Define v^k to be an arbitrary order for s_k so that $s_k = \{v^k_1, \ldots, v^k_{|s_k|}\}$. Let T^k_i be the type of v^k_i, and define the tuple type $\overline{T}^k = (\ldots((), T^k_1)\ldots, T^k_{|s_k|})$.
- Then define the array program E^k as `Apair (...Apair Anil (Avar` v^k_1`)...) (Avar` $v^k_{|s_k|}$`)` of type $\overline{T}^k$, which collects the values bound by the variables in the to-store set s_k into a (large) tuple.
- We also need a "shadow program" F^k that mirrors E^k but contains just empty values; define F^k as `Apair (...Apair Anil (`GenEmpty(T^k_1)`)...) (`GenEmpty$(T^k_{|s_1|})$`)`, which is also of type $\overline{T}^k$. GenEmpty(T) for an array tuple type T is a macro[43] producing an array program of type T, and is defined recursively as follows:
  - GenEmpty$(())$ = `Anil`
  - GenEmpty$((T_1, T_2))$ = `Apair (`GenEmpty(T_1)`) (`GenEmpty(T_2)`)`
  - GenEmpty$(t[s])$ = `Generate (Const (`ZeroValue(s)`)) (Lam i (Const (`ZeroValue(t)`))))`

  Here ZeroValue(t) for a scalar type t is the nearest equivalent of "0" for the particular type t. If t is a floating-point type or `int`, we require that ZeroValue$(t) = 0$, and furthermore we will need[44] ZeroValue(`bool`) = `false`; if the language has other single-types, we do not require any particular value. If t is $()$ the result should be $()$, and if t is (t_1, t_2) we should have ZeroValue$(t) =$ (ZeroValue(t_1), ZeroValue(t_2)). Note that for a shape type s, ZeroValue(s) is the zero shape of that type; hence, the program given for GenEmpty$(t[s])$ constructs an empty array of the given type.

Having these definitions, we can write down the output of the primal phase for the given `Acond` node. Substitute the array program `Apair a`m_1 `(Apair (`E^1`) (`F^2`))` into b_1 to obtain the program E_{then}, and substitute `Apair a`m_2 `(Apair (`F^1`) (`E^2`))` into b_2 to obtain E_{else}. Note that E_{then} and E_{else} have the same type: $(T, (\overline{T}^1, \overline{T}^2))$ where T is the type of the original `Acond` program.

The output builder is then the following:

```
Alet anall = Acond expR (E_then) (E_else)      -- 'expR' is the renamed expression
in Alet an = Afst (Avar anall)
in Alet anthen = Afst (Asnd (Avar anall))
in Alet anelse = Asnd (Asnd (Avar anall))
in Alet v^1_{|s1|} = Asnd (Avar anthen)                    -- 0 Afst constructors
in Alet v^1_{|s1|-1} = Asnd (Afst (Avar anthen))           -- 1 Afst constructor
-- ...
in Alet v^1_1 = Asnd (Afst ...(Afst (Avar anthen)...))  -- |s1| - 1 Afst constructors
in Alet v^2_{|s2|} = Asnd (Avar anelse)                    -- 0 Afst constructors
in Alet v^2_{|s2|-1} = Asnd (Afst (Avar anelse))           -- 1 Afst constructor
-- ...
in Alet v^2_1 = Asnd (Afst ...(Afst (Avar anelse)...))  -- |s2| - 1 Afst constructors
in _
```

[43] In other words, a function in the compiler, not in the compiled program.

[44] We will need this in the expression primal transform in Section 3.5.1.

The output to-store set is $s_1 \cup s_2 \cup \{\texttt{a}n\}$: any `Acond` higher up the tree will need to restore the variables from these branches again, and the result of the conditional needs to be stored as well.

This defines the primal transform for `Acond`. The intent of its builder is to first conditionally execute the correct subprogram using an `Acond`, and then to create temporaries for *both* of the branches, storing the computed values in the temporaries for the taken branch and storing empty arrays in the temporaries for the untaken branch. This ensures that only the code that the original program would execute actually gets executed (this is important to ensure that the primal-transformed program does not crash[45] if the original program does not), while simultaneously providing the temporaries of whatever branch was executed to the code generated by the future dual phase. Because we cannot conditionally bind a variable in the environment in a lexically-scoped language like Idealised Accelerate, we must bind all the temporaries to *some* value, including those of the branch not executed. This leads to the construction defined above.

The result contains a potentially large amount of let-bindings and tuple projection operators, which may raise concerns about efficiency. However, the intended target application of this algorithm is in Accelerate, where tuples of arrays are transparently converted to code dealing solely with individual arrays. This means that the let-bindings and tuple projections just result in extra book-keeping work in the compiler, but no additional runtime cost.

Another source of concern regarding efficiency may be the construction of the large tuples at the end of generated then- and else-branches of the conditional. While this does generate some management work at runtime, no full arrays are actually copied, meaning that for large enough data the overhead should be minimal. The empty arrays created also result in some constant amount of work per array; in the end, any extra runtime work is on the order of $|s_1| + |s_2|$, which is independent of the actual size of the arrays being dealt with.

Example. Consider the following example program of type $\texttt{float}[((), \texttt{int})] \to \texttt{float}[()]$:

```
1 Alam arg (Alet b = Map₁ (Lam x (PrimApp Mul (Pair (Evar x) (Const 3.0)))) (Avar₂ arg)
2           in Sum₃ (Map₄ (Lam x (PrimApp Mul (Pair (Evar x) (Index b (PrimApp Round (Evar x))))))
3                         (Avar₅ b)))
```

Or in concrete Accelerate syntax:

```
1 \arg -> let b = map (*3) x in sum (map (\x -> x * b ! round x) b)
```

This array function computes $\sum_i \texttt{b}_i \cdot \texttt{b}_{\text{round}(\texttt{b}_i)} = \sum_i 3 \cdot \texttt{arg}_i \cdot 3 \cdot \texttt{arg}_{\text{round}(3 \cdot \texttt{arg}_i)}$, where the sum ranges from $((), 0)$ to $((), n-1)$ and n is the length of `arg`.[46]

The recursive traversal starts at the `Alet` and, via the `Map` (node 1), arrives at node 2. This is a variable reference to the argument, meaning the produced builder is `Alet a2 = Avar arg in _`. The `Map` then adds its builder from Table 1 that defines `a1res`, `a1` and `a1tmp`.

Going into the body of the `Alet` we acquire a binding in the variable environment that maps `b` to label 1, hence when visiting the $\texttt{Avar}_5$ `b` via the `Sum`, we use the second `Avar` row in Table 1 to get the builder `Alet a5 = Avar a1 in _`. The second `Map`, node 4, then adds its own builder that defines `a4res`, `a4` and `a4tmp`. Finally, the `Sum` adds its own builder from the table: `Alet a3 = Sum (Avar a4) in _`. Let λ_1 be the expression function in the `Map` with label 1 in the example source code above, and similarly let λ_4 be the expression function in node 4. Let $\mathcal{P}(\lambda_1)$ and $\mathcal{P}(\lambda_4)$ be their primal lambdas from expression AD. The full builder from the primal phase for this program then looks as follows:

[45] Crashing here means "performing an out-of-bounds array indexing operation".

[46] The example programs in the algorithm description are at times very artificial. This is because they are chosen to be small while simultaneously exhibiting the features that we want to explain. The algorithm also works on more useful programs, see Section 5.
We will return to this particular example program in Section 3.4.6.

```
Alet a2 = Avar arg
in Alet a1res = Map P(λ1) (Avar a2)
in Alet a1 = Map (Lam x (Fst (Evar x))) (Avar a1res)
in Alet a1tmp = Map (Lam x (Snd (Evar x))) (Avar a1res)
in Alet a5 = Avar a1
in Alet a4res = Map P(λ4) (Avar a5)
in Alet a4 = Map (Lam x (Fst (Evar x))) (Avar a4res)
in Alet a4tmp = Map (Lam x (Snd (Evar x))) (Avar a4res)
in Alet a3 = Sum (Avar a4)
in _
```

Example with a conditional. To illustrate how the transformation of an `Acond` node looks in practice, consider the following example program:

```
Alam arg (Sum1 (Acond2 (PrimApp Lt (Pair (Index arg (Pair Nil (Const 0))) (Const 0.0)))
                       (Backpermute3 (Pair Nil (Const 5))
                                     (Lam idx (Evar idx))
                                     (Avar4 arg))
                       (Map5 (Lam x (PrimApp Mul (Pair (Const 2.0) (Evar x))))
                             (Avar6 arg))))
```

This program computes the sum of the first 5 elements of the vector argument if the argument's first element is less than zero, or the sum of the entire array with elements multiplied by two otherwise.

Via the `Sum` and the `Acond` we first transform the `Backpermute` and its `Avar` argument to produce the following builder with to-store set $\{\texttt{a3}, \texttt{a4}\}$, analogous to the previous example:

```
Alet a4 = Avar arg
in Alet a3 = Backpermute (Pair Nil (Const 5)) (Lam idx (Evar idx)) (Avar a4)
in _
```

The else-branch of the conditional is similarly transformed to produce this builder with to-store set $\{\texttt{a5}, \texttt{a5tmp}, \texttt{a6}\}$:

```
Alet a6 = Avar arg
in Alet a5res = Map P(Lam x (PrimApp Mul (Pair (Const 2.0) (Evar x)))) (Avar a6)
in Alet a5 = Map (Lam pair (Fst (Evar pair))) (Avar a5res)
in Alet a5tmp = Map (Lam pair (Snd (Evar pair))) (Avar a5res)
in _
```

Let the type of the argument `arg` be $f[((), \texttt{int})]$ for some floating-point type f. For the conditional, we arbitrarily choose $v^1 = (\texttt{a3}, \texttt{a4})$ and $v^2 = (\texttt{a5}, \texttt{a5tmp}, \texttt{a6})$. For the large tuple types, we get $\overline{T}^1 = (((), f[((), \texttt{int})]), f[((), \texttt{int})])$ and $\overline{T}^2 = ((((), f[((), \texttt{int})]), t^5_{\text{tmp}}[((), \texttt{int})]), f[((), \texttt{int})])$, where t^5_{tmp} is the t_{tmp} resulting from expression AD on the lambda in the `Map` node.

The E^k programs look as follows:

```
E1 = Apair (Apair Anil (Avar a3)) (Avar a4)
E2 = Apair (Apair (Apair Anil (Avar a5)) (Avar a5tmp)) (Avar a6)
```

To be able to construct F^2, we need a zero-like value of type t^5_{tmp} to instantiate the ZeroValue(t^5_{tmp}) macro used in the definition of F^k. In order to abstract over what t^5_{tmp} is, we leave the macro as-is in the concerned field of F^2:

```
F1 = Apair (Apair Anil
                  (Generate (Const ((), 0)) (Lam i (Const 0.0))))
           (Generate (Const ((), 0)) (Lam i (Const 0.0)))
F2 = Apair (Apair (Apair Anil
                         (Generate (Const ((), 0)) (Lam i (Const 0.0))))
                  (Generate (Const ((), 0)) (Lam i (Const (ZeroValue(t^5_tmp))))))
           (Generate (Const ((), 0)) (Lam i (Const 0.0)))
```

Using these in the final builder of the `Acond`, we arrive at the following result builder of the primal transform of the example program:

```
-- Shorthands:
--   EMPTYVF = Generate (Const ((), 0)) (Lam i (Const 0.0))                  [empty vector of float]
--   EMPTYVTMP = Generate (Const ((), 0)) (Lam i (Const (ZeroValue(t5_tmp)))) [empty vector of tmp]
Alet a2all =
        Acond (PrimApp Lt (Pair (Index arg (Pair Nil (Const 0))) (Const 0.0)))
           (Alet a4 = Avar arg
            in Alet a3 = Backpermute (Pair Nil (Const 5)) (Lam idx (Evar idx)) (Avar a4)
            in Apair a3 (Apair (Apair (Apair Anil (Avar a3)) (Avar a4))
                               (Apair (Apair (Apair Anil EMPTYVF) EMPTYVTMP) EMPTYVF)))
           (Alet a6 = Avar arg
            in Alet a5res = Map P(Lam x (PrimApp Mul (Pair (Const 2.0) (Evar x)))) (Avar a6)
            in Alet a5 = Map (Lam pair (Fst (Evar pair))) (Avar a5res)
            in Alet a5tmp = Map (Lam pair (Snd (Evar pair))) (Avar a5res)
            in Apair a5 (Apair (Apair (Apair Anil EMPTYVF) EMPTYVF)
                               (Apair (Apair (Apair Anil (Avar a5)) (Avar a5tmp)) (Avar a6))))
in Alet a2 = Afst a2all
in Alet a2then = Afst (Asnd (Avar a2all))
in Alet a2else = Asnd (Asnd (Avar a2all))
in Alet a4 = Asnd (Avar a2then)
in Alet a3 = Asnd (Afst (Avar a2then))
in Alet a6 = Asnd (Avar a2else)
in Alet a5tmp = Asnd (Afst (Avar a2else))
in Alet a5 = Asnd (Afst (Afst (Avar a2else)))
in -- now the Acond is done, and we finish with node 1, the Sum
   Alet a1 = Sum a2
in _
```

3.4.4 Dual

In the previous section we described the primal transform, which creates a builder that computes the same result as the original function did while also storing all intermediate values in variables. If we would substitute the array program `Avar a`n in this primal builder, where n is the label of a node in the program, we would create an array program that computes only the value of that node, and throws away the rest. In particular, if we take for n the label of the top-level node of the program, the resulting program would be semantically equivalent to the original program.

However, what we want to compute is not an intermediate value of the function, but the gradient of the function. In order to do this, the remainder of the algorithm consists of two components. In the first, much larger component, we compute the adjoints of all AST nodes using a recursive traversal; this recursive traversal is similar to that in the primal phase, and produces a builder. This component is described in this section. In the second component we compute the array program that, when substituted in the hole of the dual builder, produces the gradient of the global function argument; this program is the *gradient collector*. In Section 3.4.5 we describe the gradient collector and how to combine the primal and dual results into the final gradient program.

This section is organised in the following parts, indicated with paragraph headers:

- First, we define the input and output values of the recursive traversal.
- Then, with this knowledge, we describe the high-level structure and intent of the algorithm.
- As an interlude, we give a definition of the GenZero macro that will be used below.
- With this definition, we can write down the recipe for all but the most special constructors in the AST. Most of this will be specified by the four-step recipe and Table 2, but the gaps that are left open, as well as the reasoning behind the values in Table 2, will be explained individually for each AST constructor.

- Afterwards, we discuss how to transform let-bindings, which have not been covered in the previous point because they work differently.
- Finally, we describe the algorithm for the most complicated constructor, `Acond`, which introduces conditional execution.

Recursive traversal. The dual recursive traversal is similar in overall structure to the recursive traversal of the primal phase described in Section 3.4.3, but some of the details are different.[47]

The recursive algorithm takes as input three things:

- The AST of an array program of production *arr* from Fig. 1.
- An array variable environment mapping variable names to labels.
- A mapping from labels to lists of array programs, called the *contribution map*. This map collects adjoint contributions to other nodes in the program; these other nodes can be argument arrays, right-hand sides of let-bindings referred to by a variable, or the global function argument in case of an argument reference.

The initial environment at the top level contains only a mapping from the `Alam` argument to a special value identifying the global function argument. The top-level contribution map is $\{n \Rightarrow [\texttt{Generate Nil (Lam i (Const 1.0))}]\}$, where n is the label of the root node of the program; this ensures that when the root node is traversed by the algorithm, it can retrieve its adjoint (which is 1) from the contribution map in the same way as the rest of the AST nodes can.

The output of the algorithm also consists of three components:

- A builder for production *arr*.
- A set of array variable names called the *to-store* set, containing the variables used in the updated contribution map (see the next bullet point) that must be retained by a possible containing `Acond`. Compared to the primal phase, in both cases we use the to-store sets to be able to handle conditionals correctly; the difference is that in the primal phase, this set contained the temporaries necessary in the dual phase, while here we only need to ensure that the entries in the contribution map do not suddenly contain out-of-scope variable references when we cross out of a conditional branch.
- An updated contribution map.

Compared to the primal phase, the contribution map is a new input and output value. The map is threaded through the algorithm, and is the channel via which all adjoint contributions are communicated. This includes the contributions to direct arguments of nodes, as explained below.

High-level structure. The idea of the algorithm is the following: each node in the tree retrieves its own adjoint from the contribution map, stores it in its builder, and adds the adjoints of its arguments to the contribution map given to the argument nodes.

As stated above, the contribution map binds the label of a node to a *list* of array programs. In a program without let-bindings, every node receives exactly one contribution: the top node gets a zero-dimensional array with value 1, and every other node gets a contribution from its parent node. The entry in the contribution map created for the argument of a node will, therefore, always contain a list with exactly one element.

However, this is not true for the right-hand side of a let-binding, due to the introduced sharing. Each variable reference that refers to this right-hand side gets an adjoint, and each of these adjoints

[47] For an example of running the dual algorithm on a simple program, see Section 3.4.6.

is on its turn a contribution to the adjoint of the right-hand side. All of these contributions must be collected and summed (elementwise) to form the adjoint of the right-hand side. This summing comes from the principle in reverse AD, already mentioned in Section 2.1.2, that fanout in the computation graph (sharing of a value) corresponds to addition in the gradient program. Fortunately, all variable references are explicit in the AST, so we can easily see where these contributions will come from. Since Idealised Accelerate lacks operations for executing array subprograms multiple times (due to the absence of higher-order operations and looping constructs), variable references cannot be duplicated, and each `Avar` node in the program produces exactly one adjoint contribution to the right-hand side it refers to.

The only exception to this rule is conditional execution: when written in the branch of an `Acond`, an `Avar` node can be skipped entirely depending on the condition's value at runtime. We address this problem by making a non-executed `Avar` node produce an *empty* contribution in the form of an empty array. When adding contributions together, we skip empty arrays, thereby ensuring that at runtime, we only add the contributions from nodes that were actually executed.

The attentive reader notes that indicating non-execution using an empty contribution array is in-band signalling: an empty array may very well be a valid contribution from an executed node, if the target node of the contribution happened to produce an empty array in the primal phase. However, this is not a problem and we need not distinguish between a contribution that is empty because the primal array was already empty, and one that is empty because the variable reference was not executed. This is because the elementwise sum of zero or more empty arrays is always an empty array, regardless of whether any of the summands was to be included or not.

Zero arrays. Before the full description of the algorithm, we first lead with a definition. In the algorithm we will need to generate zero-filled arrays of particular sizes, for example in the `Permute` generated for array indexing or when the contribution list for a particular adjoint is empty. To do this, we define the macro GenZero(`arr`) for an array program `arr`. This macro returns an array program that is of the same shape and size as `arr`, but contains zero values (in the sense of ZeroValue as defined in Section 3.4.3 on page 47). It is defined as follows by case-analysis on the type of `arr`:

- For type $()$, define GenZero(`arr`) = `Anil`.
- For type (T_1, T_2), define GenZero(`arr`) = `Apair` (GenZero(`Afst arr`)) (GenZero(`Asnd arr`)).
- For type $t[s]$, define GenZero(`arr`) as the following:
  `Alet v = arr in Generate (Shape v) (Lam i (Const (`ZeroValue(t)`)))`, where `v` is a fresh variable with a unique prefix. We store the array program in a variable to be able to refer to it from a `Shape` call in an expression.

Note that while GenZero may recompute its argument `arr` many times, in practice it will only be called on array programs consisting only of `Avar`, `Afst` and `Asnd` nodes; in an implementation, these can be optimised to do (almost) no actual runtime work.

Recipe for most nodes. For nodes that are not `Alet`, `Acond`, `Scanl`, `Scanr` and `Permute`, the general recipe for the algorithm is as follows. Let the current node's label be n, and assume it has k array arguments (arguments of production arr) of which the labels are $m_1, \ldots, m_k$.

1. Look up the entry in the contribution map under key n, which is a list L of array programs. If there is no such entry (possible if a let-bound variable is not referenced at all), set $L = []$, the empty list. The adjoint of the current node is then the array program EltwiseEIsum(L, `a`n). Here, EltwiseEIsum ("elementwise empty-ignoring sum") is a macro taking a list L of array programs, all of some type T, and a reference array program of type T, and is defined by case distinction on T:

- If T is (), then EltwiseEIsum(L, `ref`) = `Anil`.
- If T is (T_1, T_2), then EltwiseEIsum(L, `ref`) = `Apair (`EltwiseEIsum(L'_1, `Afst ref`)`) (`EltwiseEIsum(L'_2, `Asnd ref`)`)`, where L'_1 is the list L with `Afst` applied to each element, and L'_2 analogously with `Asnd`. To prevent recomputation of expensive array programs, the programs in L can first be bound to variables using `Alet` nodes.
- If T is $t[s]$, then EltwiseEIsum([], `ref`) = GenZero(`ref`), and for non-empty lists L, we `Generate` an array with size (`Shape ref`) and with the element at index `i` given by the sum of the elements of the arrays in L at index `i`, replacing an element by ZeroValue(t) if the index is out-of-bounds in that particular array. This conditional can be effected using a `Cond` node. The sum of two scalar values is elementwise over tuples; the sum of non-numeric values may be an arbitrary valid value of the relevant single-type.

Note that we cannot just combine the adjoint contributions using a `ZipWith` node, as in `ZipWith (Lam p (PrimApp Add p))`: firstly, we have to explicitly ignore empty contributions, but secondly, the result (and thus the adjoint) of `ZipWith` nodes has as shape the elementwise minimum of the shapes of its argument (see Section 3.1). Hence, contributions generated by a `ZipWith` may not have the full required size for the adjoint of its arguments. To fix this, we explicitly use the `Generate` construction described above.

2. Determine the contribution map to be passed to the array program arguments of the current node. This is the contribution map received by the current node, plus the additions listed in the row in Table 2 in the "Contribution map additions" column. If both maps contain an entry for the same key, the corresponding lists are concatenated.

   Optionally, one can remove the entry for the current node from the contribution map in this stage. This is optional, since the entry will not be inspected or used ever again, but for the same reason leaving it in may be a waste of memory.

   Table 2 contains some new notation, and some entries are incomplete. "λ`fst`" stands for `Lam x (Fst (Evar x))` (analogously "λ`snd`"), and (`Lam x e1`)$\circ$(`Lam y e2`) stands for `Lam y (Let x = e1 in e2)`, as in Table 1. The incomplete entries, as well as the "IA" notation, will be explained below when we discuss each of the contructors separately.

   For example, if the current node is `Apair`$_2$ `arr3`$_1$ `arr4`$_4$ and the contribution map received by the current node is $\{1 \Rightarrow [$`arr1`$], 2 \Rightarrow [$`arr2`$]\}$, the contribution map to be passed to the arguments in the next step is $\{1 \Rightarrow [$`arr1, Afst (Avar d2)`$], 4 \Rightarrow [$`Asnd (Avar d2)`$]\}$. (Here we opted to remove the entry for the current node.) The adjoint for the current node (step 1) is `arr2` in this example.

3. Recurse on the k array arguments. All invocations get the same variable environment, which is the variable environment received by the current node. For the contribution map, invocation 1 gets the map computed in step 2, and invocation $i \in \{2, \ldots, k\}$ gets the contribution map returned from invocation $i - 1$.

4. Let b_{args} be the composition of the builders returned from the recursive invocations (in any order), and let b be the builder from Table 2 (second column), where "`@`" denotes the adjoint computed in step 1. The output builder for the current node is then $b \circ b_{\text{args}}$.[48]

   The output contribution map from the current node is the map returned from invocation k in step 3. The output to-store set is the union of the to-store sets from the recursive

[48] Recall that in the primal phase, the current node's builder was to be composed *after* those of the arguments. This reversal of composition order is the point where we "execute the program backwards" in the dual pass.

invocations, together with the to-store set additions in Table 2 (fourth column); this union will always be disjoint.

As stated, the above recipe does not fully define the dual transform even for the constructors that it does cover (i.e. all except `Alet` and `Acond` and the unsupported constructors `Scanl`, `Scanr`, `Permute`), because some of the notation used in their builders and contribution map additions is yet undefined. Therefore, we will discuss each of the constructors in Table 2 individually below, filling in these gaps and explaining the reasoning behind the definitions. The `Alet` and `Acond` nodes are discussed separately afterwards.

- `Use` and `Anil` nodes receive an adjoint as usual, but since their adjoint on its own is not interesting, and they also do not have any arguments to contribute to, we need not store it. Hence their builder is empty and they create no additional contribution map entries.
- `Avar` nodes have no direct arguments, but do indirectly refer to the right-hand side their variable is bound to. The adjoint of the node is stored in the `d`n variable, to be used in the contribution it places in the contribution map. Since we want `d`n to persist outside of the current branch of any `Acond`s that contain the current node (since the binding `Alet` or `Alam` may be arbitrarily far up the AST), we add `d`n to the to-store set.
- The adjoints of the arguments to an `Apair` node are the two components of the adjoint of the `Apair` itself. Hence, we add them to the contribution map at the correct keys.

  Note that the to-store set for an `Apair` node is empty, even though we do create contribution map additions containing variable references. The key point is that these contribution map entries will be used only in the builders produced by the direct arguments to the `Apair` node, which are composed with the builder for the current `Apair` node: there is no chance for the `d`n variable to get lost in-between. This is in contrast to an `Avar` node as discussed in the previous point, because there the contribution map entry will only be read at the point where the variable is bound, which may cross any number of `Acond` nodes.

  The same reasoning about why the to-store set does not contain `d`n holds for all subsequent nodes in this list; we will not repeat the reasoning there.
- The argument of an `Afst` node is of pair type; the adjoint of the left half is evidently the adjoint of the `Afst` node itself, but the right half is completely ignored by the `Afst`. Hence, the right half gets a zero adjoint, as computed by GenZero, which uses the primal value of the right half of the argument to get the right shapes. `Asnd` is analogous to `Afst`, swapping all sides.
- Assuming the target shape has the same number of elements as the source shape, `Reshape` is just a permutation of the elements in its argument array. Hence, the inverse permutation (a `Reshape` to the shape of its original argument) applied on the adjoint of the result is the adjoint of the argument array.
- Fanout translates to addition in the dual, and this holds the same for `Replicate`. Its derivative is best explained using an example; the general form follows by extrapolation.

  Consider the array program `Replicate exp arr` where `arr` is an array program resulting in an array of size $(((),2),3)$ and `exp` is an expression resulting in the value $(((((),4),()),5),())$. If `r` is the result of `Replicate exp arr`, then the size of `r` is $(((((),4),2),5),3)$ and $\texttt{r}_{((((),i),j),k),\ell)} = \texttt{arr}_{(((),j),\ell)}$.

  For the dual, suppose that the label of the `Replicate` node is 1 and thus its adjoint is stored in `d1`, of size $(((((),4),2),5),3)$. We want to obtain the adjoint `da` (size $(((),2),3)$) of `arr` where $\texttt{da}_{(((),j),\ell)} = \sum_{i=1}^{4}\sum_{k=1}^{5}\texttt{dr}_{((((),i),j),k),\ell)}$. We can do this with the following program:

arr primitive	Builder	Contribution map additions	To-store set add.
$\texttt{Use}_n$ lit / $\texttt{Anil}_n$	_	{}	{}
$\texttt{Avar}_n$ argument	Alet dn = @ in _	{argument ⇒ [Avar dn]}	{dn}
$\texttt{Avar}_n$ $\texttt{var}_{\to m}$	Alet dn = @ in _	{m ⇒ [Avar dn]}	{dn}
Alet a = rhs in body	_	{}	{}
$\texttt{Apair}_n$ $\texttt{arr1}_{m_1}$ $\texttt{arr2}_{m_2}$	Alet dn = @ in _	{m_1 ⇒ [Afst (Avar dn)], m_2 ⇒ [Asnd (Avar dn)]}	{}
$\texttt{Afst}_n$ $\texttt{arr}_m$	Alet dn = @ in _	{m ⇒ [Apair (Avar dn) (GenZero(Asnd (Avar am)))]}	{}
$\texttt{Asnd}_n$ $\texttt{arr}_m$	Alet dn = @ in _	{m ⇒ [Apair (GenZero(Afst (Avar am))) (Avar dn)]}	{}
$\texttt{Reshape}_n$ exp $\texttt{arr}_m$	Alet dn = @ in _	{m ⇒ [Reshape (Shape am) (Avar dn)]}	{}
$\texttt{Replicate}_n$ exp $\texttt{arr}_m$	Alet dn = @ in _	{m ⇒ [Sum (Reshape ... (Backpermute ... ... (Avar dn)))]} (Some parts elided; see text.)	{}
$\texttt{Slice}_n$ exp $\texttt{arr}_m$	Alet dn = @ in _	{m ⇒ [Generate (Shape am) ...]} (Some parts elided; see text.)	{}
$\texttt{Generate}_n$ exp fun	Alet dn = @ in Alet d$n$ia = Map λsnd (ZipWith $\mathcal{D}$(fun) (Avar dn) (Avar a$n$tmp)) in _	IA(fun, d$n$ia)	{d$n$ia}
$\texttt{Map}_n$ fun $\texttt{arr}_m$	Alet dn = @ in Alet d$n$res = ZipWith $\mathcal{D}$(fun) (Avar dn) (Avar a$n$tmp) in Alet d$n$ia = Map λsnd (Avar d$n$res) in _	{m ⇒ [Map λfst (Avar d$n$res)]} ∪ IA(fun, d$n$ia)	{d$n$ia}
$\texttt{ZipWith}_n$ fun $\texttt{arr}_{m_1}$ $\texttt{arr}_{m_2}$	Alet dn = @ in Alet d$n$res = ZipWith $\mathcal{D}$(fun) (Avar dn) (Avar a$n$tmp) in Alet d$n$ia = Map λsnd (Avar d$n$res) in _	{m_1 ⇒ [Map λfst∘λfst (Avar d$n$res)] m_2 ⇒ [Map λsnd∘λfst (Avar d$n$res)]} ∪ IA(fun, d$n$ia)	{d$n$ia}
$\texttt{Sum}_n$ $\texttt{arr}_m$	Alet dn = @ in _	{m ⇒ [Backpermute (Shape am) λfst (Avar dn)]}	{}
$\texttt{Foldl}_n$ fun exp $\texttt{arr}_m$ (where arr has type $f[s]$ for a floating-point type f, and fun and exp contain no Index nodes)	Alet dn = @ in Alet dns = INIT (Scanl fun exp (Avar am)) in Alet dnd = ZipWith λfst∘$\mathcal{D}$(fun)∘$\mathcal{P}$(fun) (Avar dns) (Avar am) in Alet d$n$tmp = ZipWith (Lam p (PrimApp Mul p)) (Map λsnd (Avar dnd)) (TAIL (Scanr (Lam p (PrimApp Mul p)) (Const 1) (Map λfst (Avar dnd)))) in _ (For definitions of INIT and TAIL, see the text.)	{m ⇒ [ZipWith (Lam p (PrimApp Mul p)) (Avar dn) (Backpermute (Shape am) (Lam i (Fst (Evar i))) (Avar d$n$tmp))]}	{}
$\texttt{Backpermute}_n$ exp (Lam x body) $\texttt{arr}_m$	Alet dn = @ in _	{m ⇒ [Permute (Lam p (PrimApp Add p)) (GenZero(am)) (Lam x (Pair (Const true) body)) (Avar dn)]}	{}
$\texttt{Acond}_n$ exp $\texttt{arr1}_{m_1}$ $\texttt{arr2}_{m_2}$	(special, see text)		
Scanl/Scanr/Permute	(not supported, see text)		

Table 2: Rules for the dual phase of array AD. "@" is a placeholder for the computed adjoint of the node itself. $\mathcal{D}$(fun) means the dual lambda for fun, and e.g. λfst∘λsnd is shorthand for Lam x (Fst (Snd (Evar x))). For the definition of the IA macro in the third column, see the text.

```
Sum (Reshape (Const ((((), 2), 3), 20))  -- 4 * 5 = 20
        (Backpermute (Const ((((), 2), 3), 4), 5))
            (Lam idx
                (Let i = Snd (Fst (Fst (Fst (Evar idx))))
                 in Let j = Snd (Fst (Fst (Evar idx)))
                 in Let k = Snd (Fst (Evar idx))
                 in Let l = Snd (Evar idx)
                 in Pair (Pair (Pair (Pair Nil (Evar j)) (Evar l)) (Evar i)) (Evar k)))
            d1))
```

In other words, we first use `Backpermute` to shuffle the "expanded" indices to the end, `Reshape` those dimensions into a single dimension with size the product of those dimensions, and then `Sum` those. The general form, including the expressions using `Shape` to obtain the values 2, 3, 4, 5 in the program above, is elided here for reasons of conciseness. The contribution map addition in the third column of Table 2 is then $\{m \Rightarrow [...]\}$, where this program goes in place of the "...".

- `Slice exp arr`$^{t[s]}$ selects one slice of certain dimensions of the shape of the argument, thereby eliminating those dimensions. A way to express its derivative is to use `Generate` with the shape of `arr`, that picks the corresponding value from the `Slice` adjoint if the sliced dimensions match the values in `exp`, and ZeroValue(t) otherwise. The conditional is expressed using a `Cond` node; for a reasonable translation we assume the availability of a binary "logical AND" operation in *op* in Fig. 1. The full contribution program is elided for conciseness.

- A `Generate` node contains an expression function that we need to differentiate using expression AD. Recalling from Section 3.4.1, the dual lambda from expression AD returns not only the gradient of the function input but also an index adjoint information tuple (IAI tuple); since in this case the input to the expression function is discrete (namely, the element index), we are only interested in the IAI tuple, which we store in `d`n`ia`. A `ZipWith` is used to supply the adjoints and the expression temporaries to the dual lambda.

  Note that the dual lambda from expression AD, written $\mathcal{D}$(`fun`) in Table 2, contains references to program variables that must be renamed to the corresponding `a`N names in the builder for `Generate`. This renaming is elided in the table for conciseness.

  A `Generate` node has no array arguments to compute the adjoint of, but any indexed variable references need their adjoints registered in the contribution map. To represent these contributions, we use the macro IA(`fun`, `ia`), taking the expression function in question (`fun`) and (a variable referring to) the array of IAI tuples (`ia`); this macro is defined in natural language as follows.[49]

  Suppose there are k `Index` nodes in `fun`, indexing array variables pointing to right-hand sides with labels $\ell_1, \ldots, \ell_k$; then each IAI tuple in `ia` contains three values, say d_i, idx_i and $exec_i$ for $i \in \{1, \ldots, k\}$ (see Section 3.4.1). The meaning of these values is that for each i, we have an adjoint d_i to be contributed to the array node with label ℓ_i at the target index idx_i if $exec_i$ is `true`. This $exec_i$ is the boolean that indicates whether the `Index` node was executed at all. Let the type of node ℓ_i be $t_i[s_i]$; note that we know this is not a tuple type because a reference to it is the argument to an `Index` node. For each i, we will define an array program $contrib_i$ that has the same shape as `a`ℓ_i and represents precisely the contributions to the adjoint of node ℓ_i resulting from the i'th `Index` node in `fun`. These contributions are put in a map to form the result of IA(`fun`, `ia`), where care must be taken to put contributions to the same node in a list instead of creating multiple colliding bindings in the map.

[49] Note that this is the central place where our reverse AD algorithm deals with array indexing.

How, then, do we write $contrib_i$? Like in Section 3.3.2, we will do this with a `Permute` node. Assume ExtractD$_i$(`exp`), ExtractIdx$_i$(`exp`) and ExtractExec$_i$(`exp`) are expressions that extract respectively d_i, idx_i and $exec_i$ from an IAI-tuple-typed expression `exp`. These three macros, together with the labels ℓ_i, are the realisation of the "information about the layout of t_{idxadj}" that we mentioned in Section 3.4.1. Following the example of Section 3.3.2, we can then write:

```
1  Permute (Lam p (PrimApp Add p))
2          (Generate (Shape aℓi) (Lam i (ZeroValue(ti))))
3          (Lam i (Let iai = Index ia i
4                  in Pair (ExtractExeci(Evar iai)) (ExtractIdxi(Evar iai)))
5          (Map (Lam iai (ExtractDi(Evar iai))) (Avar ia))
```

This is the program $contrib_i$. Note how we use the flag indicating whether the `Index` node was executed to determine whether we need to permute the corresponding adjoint. The `ia` variable used here, the array of IAI tuples, is named `d`n`ia` in the row for `Generate` in Table 2.

This finishes the definition of the IA macro, which will also be used for `Map` and `ZipWith` below.

- The `fun` argument to a `Map` may also contain array indexing, so we include the same logic using IA() as described above for `Generate`. The difference is that here the argument to `fun` is a value from an array; therefore, the first half of the result pair of the dual lambda gives the adjoint of that array, instead of being thrown away. The $\mathcal{D}$(`fun`) in Table 2 should also have its internal array variable references renamed to `a`N for the correct values of N.
- `ZipWith` is analogous to `Map`, except that there are two input arrays. The produced contributions pick the right values from the dual lambda result to create the adjoints of the argument arrays. Again, we use IA() to create the contributions arising from array indexing, and the internal array variables in the dual lambda are renamed.
- The derivative builder for `Sum` can equivalently be written using a `Replicate`, a `Backpermute` or a `Generate`; each of these primitives is a specialisation of the next one (in the sense of Section 2.3.1). We chose to give the version using `Backpermute` in Table 2 because it is most succinctly expressible; however, since the `Replicate` primitive is most specific in its functionality, it is the easiest version for the optimiser to fuse with the rest of the program, and hence that is the version that our implementation uses.
- The most complicated derivative is that for `Foldl`. The two additional shorthands here are `INIT` and `TAIL`, defined as follows:

```
INIT arr = Backpermute (Pair (Fst (Shape arr)) (PrimApp Add (Snd (Shape arr)) (Const -1)))
                       (Lam i (Evar i))
                       arr
TAIL arr = Backpermute (Pair (Fst (Shape arr)) (PrimApp Add (Snd (Shape arr)) (Const -1)))
                       (Lam i (Pair (Fst (Evar i)) (PrimApp Add (Snd (Evar i)) (Const 1))))
                       arr
```

In other words, `INIT` and `TAIL` are like the standard Haskell functions `init` and `tail` applied on the inner dimension of the array.

As stated in Section 3.2.1, the code in Table 2 does not allow for array indexing in `fun` or `exp`, and requires that the element type of the folded array is a floating-point value. This is not expected to be a fundamental limitation of the algorithm.

As with `Generate`, `Map` and `ZipWith`, internal array variable references in $\mathcal{D}$(`fun`) must be renamed to the correct `a`N names.

- Where a `Backpermute` defines the value of the target array by determining where each index *comes from* in the source array, a `Permute` does it by determining where each element in the source array *maps to* in the target array.

  This makes the derivative of a `Backpermute` easily expressible in terms of a `Permute`. Indeed, since we want to permute the adjoints of the result of the `Backpermute` back to its input array, we can re-use the index mapping function from the `Backpermute` in the `Permute`, and map each adjoint precisely to the cell where the primal value was obtained from.

  In other words, for each index i in `d`n, `body` gives where the corresponding primal value came from, and thus where we should permute its adjoint to. We never skip permuting a value (hence the `Const true`), and we add any adjoints being mapped to the same destination cell (because fanout translates to addition in the dual).

  Note that the expression function in a `Backpermute` node has a discrete result, meaning that the (hypothetical) adjoint of its result, and thus of all its internal intermediate values, is zero. This means that we can completely ignore any array indexing happening in this expression function for the purposes of AD.

  Finally, the occurrence of `body` in the contribution map addition from `Backpermute` should have its internal array variable references renamed as with `Foldl` etc. above.

Let-bindings. Suppose we are visiting an `Alet a = rhs`$_{m_1}$ `in body`$_{m_2}$ node. Let C be the received contribution map and Γ be the received variable environment. First the algorithm is recursively invoked on `body` with environment $\Gamma[\texttt{a} \coloneqq m_1]$ and contribution map C, producing builder b_{body}, to-store set s_{body} and contribution map C_{body}. We can pass the unmodified contribution map since since the "label of" the `Alet` is the label of its body, and thus the adjoint contributions to `body` are already present in the contribution map under the correct key.

Then, we traverse `rhs` with the original Γ and with contribution map C_{body}, producing builder b_{rhs}, to-store set s_{rhs} and contribution map C_{rhs}. The output builder is $b_{\text{body}} \circ b_{\text{rhs}}$, the output to-store set is $s_{\text{body}} \cup s_{\text{rhs}}$ (these two sets are always disjoint), and the output contribution map is C_{rhs}.

Conditionals. Having covered all the other nodes, the remaining question is how the dual algorithm works on nodes of the form `Acond`$_n$ `exp arr1`$_{m_1}$ `arr2`$_{m_2}$. Firstly, because the condition expression `exp` has a discrete result type, it cannot contribute any adjoints and can thus be ignored for the purposes of the dual pass, except for renaming any array variable references in it to the correct `a`N names. Let `expR` be this renamed expression.

As in the usual recipe for most nodes (page 52), we first collect the adjoint of the `Acond` using EltwiseEIsum(). We add $\{m_1 \Rightarrow \texttt{d}n, m_2 \Rightarrow \texttt{d}n\}$ to the incoming contribution map to produce C'; the adjoint of either branch is the same as the adjoint of the `Acond`. Traversing `arr1` with an unchanged variable environment and contribution map C' then produces builder b_1, to-store set s_1 and contribution map C_1. Traversing `arr2` with an unchanged variable environment and contribution map C_1 similarly produces b_2, s_2 and C_2.

Define v^k, T^k, $\overline{T}^k$, E^k and F^k based on s_1 and s_2 just like in Section 3.4.3, page 45. Let E_{then} be the array program obtained by substituting the program `Apair (`E^1`) (`F^2`)` into b_1, and let E_{else} be obtained by substituting `Apair (`F^1`) (`E^2`)` into b_2. Note that compared to the primal algorithm, we do not need to produce any particular result just for the `Acond`; all we need are the intermediate adjoints necessary for the contributions from variables in the branches, which are contained in the contribution map C_2. E_{then} and E_{else} again have the same type, namely $(\overline{T}^1, \overline{T}^2)$.

The output builder of the `Acond` is then the following, analogous to the primal case but without the separate an result.

```
Alet dnall = Acond expR (E_then) (E_else)      -- 'expR' is the renamed expression
in Alet dnthen = Afst (Avar dnall)
in Alet dnelse = Asnd (Avar dnall)
in Alet v^1_|s1|   = Asnd (Avar dnthen)                    -- 0 Afst constructors
in Alet v^1_|s1|-1 = Asnd (Afst (Avar dnthen))             -- 1 Afst constructor
-- ...
in Alet v^1_1 = Asnd (Afst ...(Afst (Avar dnthen))...)     -- |s1| − 1 Afst constructors
in Alet v^2_|s2|   = Asnd (Avar dnelse)                    -- 0 Afst constructors
in Alet v^2_|s2|-1 = Asnd (Afst (Avar dnelse))             -- 1 Afst constructor
-- ...
in Alet v^2_1 = Asnd (Afst ...(Afst (Avar dnelse))...)     -- |s2| − 1 Afst constructors
in _
```

The output to-store set is $s_1 \cup s_2$; the output contribution map is C_2.

3.4.5 Combining primal and dual

In Sections 3.4.3 and 3.4.4 we described the primal and dual phases of the array AD algorithm. Both produced a builder for production *arr* (Fig. 1): call these b_p and b_d. Recall that the input to the AD algorithm was an array function of production *afun*, and thus of the form `Alam argument body` with `body` of production *arr*. The output of the algorithm must be a new array function taking the original input argument (of type, say, T) and returning the gradient of that argument (again of type T). This output array function is `Alam argument (...)`, where the body in place of `(...)` is the builder composition $b_\mathrm{p} \circ b_\mathrm{d}$ with E_gradient substituted in the hole.

Here, E_gradient is the *gradient collector*, and is equal to EltwiseEIsum$(L, \texttt{argument})$, where L is the list from the contribution map resulting from the dual phase at the key corresponding to the global function argument. Just like in step 1 of the recipe in Section 3.4.4 on page 52, if that key is not present in the map, the empty list is used. The result is an array program that collects the gradient of the global function argument assuming that it is substituted in the dual builder.

This concludes the description of the array AD algorithm. What we have left abstract until now is the expression AD algorithm, which we will discuss in Section 3.5.

3.4.6 Example

Before concluding and moving on to expression AD, let us go through the dual phase of one of the example programs given in Section 3.4.3. We gave the following program:

```
Alam arg (Alet b = Map_1 (Lam x (PrimApp Mul (Pair (Evar x) (Const 3.0)))) (Avar_2 arg)
          in Sum_3 (Map_4 (Lam x (PrimApp Mul (Pair (Evar x) (Index b (PrimApp Round (Evar x))))))
                          (Avar_5 b)))
```

The argument `arg` is a vector of `float` values. As the primal builder (b_p), we got the following:

```
Alet a2 = Avar arg
in Alet a1res = Map P(λ1) (Avar a2)
in Alet a1 = Map (Lam x (Fst (Evar x))) (Avar a1res)
in Alet a1tmp = Map (Lam x (Snd (Evar x))) (Avar a1res)
in Alet a5 = Avar a1
in Alet a4res = Map P(λ4) (Avar a5)
in Alet a4 = Map (Lam x (Fst (Evar x))) (Avar a4res)
in Alet a4tmp = Map (Lam x (Snd (Evar x))) (Avar a4res)
in Alet a3 = Sum (Avar a4)
in _
```

where λ_1 and λ_4 are the expression functions in node 1 and node 4, respectively.

Now we will construct the dual builder (b_{d}) using the dual algorithm. Since the program does not have a conditional, we will ignore the to-store sets.

Starting at the top-level `Alet`, we immediately continue to traverse its body, node 3, which is a `Sum`. The initial contribution map is $\{3 \Rightarrow [\texttt{Generate Nil (Lam i (Const 1.0))}]\}$, which is passed on to the `Sum`, where, following the recipe on page 52, we first collect its adjoint. In this case, this is precisely the single item in the contribution map, and hence the builder for node 3 ("b" in step 4 of the recipe) is the following:

```
Alet d3 = Generate Nil (Lam i (Const 1.0)) in _
```

For the contribution map passed to its argument, node 4, we first remove the entry for node 3, and then add the addition from Table 2 to arrive at the following map:

$$\{4 \Rightarrow [\texttt{Backpermute (Shape a4) (Lam i (Fst (Evar i))) (Avar d3)}]\} \tag{2}$$

We recurse to node 4, where the adjoint is precisely the one item in the contribution map again. Looking in Table 2, the builder for node 4 is:

```
Alet d4 = Backpermute (Shape a4) (Lam i (Fst (Evar i))) (Avar d3)
in Alet d4res = ZipWith 𝒟(λ4) (Avar d4) (Avar a4tmp)
in Alet d4ia = Map λsnd (Avar d4res)
in _
```

For the contribution map to its argument, node 5, we notice that λ_4 has an array indexing operation. This means that IA(λ_4, `d4ia`) has one entry: at key 1 (the label of the right-hand side referred to by the variable `b`) it has a singleton list containing the requisite `Permute`. Suppose that, as in Section 3.4.1, the element type of `d4ia` (i.e. t_{idxadj} for λ_4) is organised as ((), ((adjoint, target index), was-executed)). Then the contribution is the following, where we have already filled in ExtractExec() and similar.

```
Permute (Lam p (PrimApp Add p))
        (Generate (Shape a1) (Lam i (Const 0.0)))
        (Lam i (Let iai = Index d4ia i
                in Pair (Snd (Snd (Evar iai))) (Snd (Fst (Snd (Evar iai))))))
        (Map (Lam iai (Fst (Fst (Snd (Evar iai))))) (Avar d4ia))
```

Note the `0.0` from ZeroValue(`float`). To recognise the expansion of ExtractExec() and similar, compare this program with the template in the description of the dual of `Generate` in Section 3.4.4 (page 56), where the IA() macro was defined.

Knowing the expansion of IA(λ_4, `d4ia`), we can construct the contribution map for node 5 from step 2 of the recipe for node 4. This is the received contribution map (Eq. (2)) minus the value for key 4 (resulting in the empty map), plus the additions from the `Map` row of Table 2. This results in the following map, where "`Permute ...`" is short for the 5-line program listed above:

$$\{5 \Rightarrow [\texttt{Map (Lam i (Fst (Evar i))) (Avar d4res)}], 1 \Rightarrow [\texttt{Permute ...}]\}$$

In recipe step 3 of node 4, we recurse further to node 5, which adds $\{1 \Rightarrow [\texttt{Avar d5}]\}$ to the received contribution map (producing a list at key 1 with two items), and produces the builder `Alet d5 = Map λfst (Avar d4res) in _` by using the value at key 5 in the received map. This propagates upward in step 4 of the recipe for nodes 4 and 3 so that the output contribution map of node 3 (the `Sum`) is $\{1 \Rightarrow [\texttt{Permute ...}, \texttt{Avar d5}]\}$ and the output builder of node 3 is the following:

```
Alet d3 = Generate Nil (Lam i (Const 1.0))                          -- from node 3
in Alet d4 = Backpermute (Shape a4) (Lam i (Fst (Evar i))) (Avar d3)  -- from node 4
in Alet d4res = ZipWith 𝒟(λ4) (Avar d4) (Avar a4tmp)                -- from node 4
in Alet d4ia = Map (Lam i (Snd (Evar i))) (Avar d4res)              -- from node 4
in Alet d5 = Map (Lam i (Fst (Evar i))) (Avar d4res)                -- from node 5
in _
```

Note that the `Permute` only shows up in the contribution map and not yet in the builder.

Having completed the traversal of the body of the `Alet`, we continue with its right-hand side: node 1. In step 1 of the recipe for node 1, we collect its adjoint from the contribution map; however, the relevant list now contains two programs. This means that we must take the elementwise sum of these programs using the EltwiseEIsum() macro as explained in the recipe. Inlining the result at the "`@`" in the builder for the `Map` that is node 1, we get the following builder:

```
Alet d1 =
    Alet d1a1 = Permute (Lam p (PrimApp Add p))
                        (Generate (Shape a1) (Lam i (Const 0.0)))
                        (Lam i (Let iai = Index d4ia i
                                in Pair (Snd (Snd (Evar iai))) (Snd (Fst (Snd (Evar iai))))))
                        (Map (Lam iai (Fst (Fst (Snd (Evar iai))))) (Avar d4ia))
    in Alet d1a2 = Avar d5
    in Generate (Shape a1)
                (Lam i
                   (Let x1 = Cond (PrimApp Lt (Snd (Evar i)) (Snd (Shape d1a1)))
                                  (Index d1a1 i)
                                  (Const 0.0)
                    in Let x2 = Cond (PrimApp Lt (Snd (Evar i)) (Snd (Shape d1a2)))
                                     (Index d1a2 i)
                                     (Const 0.0)
                    in x1 + x2))
in Alet d1res = ZipWith 𝒟(λ1) (Avar d1) (Avar a1tmp)
in Alet d1ia = Map (Lam i (Snd (Evar i))) (Avar d1res)
in _
```

Here `d1a1` and `d1a2` are the two contributions to the adjoint, and we use a `Generate` as specified in the definition of EltwiseEIsum() to add these arrays together. In this case, we know that the two arrays always have the same size, so we could also just have assigned the program `ZipWith (Lam p (PrimApp Add p)) (Permute ...) (Avar d5)` to `d1`. However, in general, we get something of the form shown above. Note also the `0.0` in the `Generate` and the addition of `x1` and `x2`, which respectively come from ZeroValue(`float`) and the tuple-elementwise addition specified in the definition of EltwiseEIsum(). Note furthermore that we assigned the two contribution arrays to variables beforehand; this is necessary to be able to refer to them from the expression in the `Generate`, although in this particular case we could have used `d5` directly instead of going through the separate `d1a2` variable.

Because λ_1 does not have array indexing, we have $\mathrm{IA}(\lambda_1, \texttt{d1ia}) = \{\}$, and therefore contribution map that node 1 passes to its argument (node 2) is simply $\{2 \Rightarrow [\texttt{Map } \lambda\texttt{fst } (\texttt{Avar d1res})]\}$. In node 2 we produce the builder `Alet d2 = Map (Lam i (Fst (Evar i))) (Avar d1res) in _` and the contribution map:

$$\{\texttt{arg} \Rightarrow [\texttt{Avar d2}]\} \tag{3}$$

This propagates up through node 1 to become the output contribution map from node 1; the output builder of node 1 is the 19-line builder above composed with the single-`Alet` builder from node 2.

The output contribution map from the `Alet` at the top level is the output contribution map from node 1, which is the map from node 2 shown in Eq. (3); this means that the gradient collector E_{gradient}, as described in Section 3.4.5, is simply `Avar d2`. This is because the adjoint contribution list for the global function argument contains precisely one entry.

The builder produced by the `Alet` at the top level is the composition of the builder of its body (the output builder of node 3, say b_3) and the builder of its right-hand side (the output builder of node 1, say b_1). Putting E_{gradient} in the hole of $b_\text{p} \circ b_\text{d} = b_\text{p} \circ b_3 \circ b_1$, and wrapping the resulting program in the required `Alam`, we get the following gradient function:

```
Alam arg
  (Alet a2 = Avar arg                                                    -- primal...
   in Alet a1res = Map P(λ1) (Avar a2)
   in Alet a1 = Map (Lam x (Fst (Evar x))) (Avar a1res)
   in Alet a1tmp = Map (Lam x (Snd (Evar x))) (Avar a1res)
   in Alet a5 = Avar a1
   in Alet a4res = Map P(λ4) (Avar a5)
   in Alet a4 = Map (Lam x (Fst (Evar x))) (Avar a4res)
   in Alet a4tmp = Map (Lam x (Snd (Evar x))) (Avar a4res)
   in Alet a3 = Sum (Avar a4)                                            -- end of primal
   in Alet d3 = Generate Nil (Lam i (Const 1.0))                         -- from node 3
   in Alet d4 = Backpermute (Shape a4) (Lam i (Fst (Evar i))) (Avar d3)  -- from node 4
   in Alet d4res = ZipWith D(λ4) (Avar d4) (Avar a4tmp)                  -- from node 4
   in Alet d4ia = Map (Lam i (Snd (Evar i))) (Avar d4res)                -- from node 4
   in Alet d5 = Map (Lam i (Fst (Evar i))) (Avar d4res)                  -- from node 5
   in Alet d1 =                                                          -- from node 1
       Alet d1a1 = Permute (Lam p (PrimApp Add p))
                           (Generate (Shape a1) (Lam i (Const 0.0)))
                           (Lam i (Let iai = Index d4ia i
                                   in Pair (Snd (Snd (Evar iai))) (Snd (Fst (Snd (Evar iai))))))
                           (Map (Lam iai (Fst (Fst (Snd (Evar iai))))) (Avar d4ia))
       in Alet d1a2 = Avar d5
       in Generate (Shape a1)
                   (Lam i
                      (Let x1 = Cond (PrimApp Lt (Snd (Evar i)) (Snd (Shape d1a1)))
                                     (Index d1a1 i)
                                     (Const 0.0)
                       in Let x2 = Cond (PrimApp Lt (Snd (Evar i)) (Snd (Shape d1a2)))
                                        (Index d1a2 i)
                                        (Const 0.0)
                       in x1 + x2))
   in Alet d1res = ZipWith D(λ1) (Avar d1) (Avar a1tmp)                  -- from node 1
   in Alet d1ia = Map (Lam i (Snd (Evar i))) (Avar d1res)                -- from node 1
   in Alet d2 = Map (Lam i (Fst (Evar i))) (Avar d1res)                  -- from node 2
   in Avar d2)                                                           -- E_gradient
```

Inserting the correct primal and dual lambdas from expression AD for the $\mathcal{P}$ and $\mathcal{D}$ notations in the above program gives the result. Notable is that, as expected, the array `d1ia` (which contains no useful information, since λ_1 contains no array indexing) is unused in the program, and can hence be optimised away using dead-code elimination.

3.4.7 Conclusions

When applying the array AD algorithm described in the sections above on an array function F, there are two places where we create program fragments that use types that did not already occur in F. Firstly that is in expression AD, from which we get a t_{tmp} and a t_{idxadj}, which are tuples of pre-existing types, but were not themselves used in F yet (most likely). Secondly that is in the handling of conditionals: there we use the to-store sets to construct a tuple type containing the types of intermediate expressions in the branches. Here, too, the constituent types of the tuple did already exist in F, but the full type most likely did not.

As we will see in Section 3.5, the construction of the large tuple in the handling of `Acond` really proceeds in the same way as the creation of t_{tmp} in expression AD. Nevertheless, the observation remains that the places where the produced gradient program is "not like F" are when handling the construct that makes (Idealised) Accelerate second-order (embedding of expressions in array programs), and when handling dynamic control flow.

It can be expected that these are the harder parts of designing a reverse AD algorithm. Performing reverse AD on a first-order language, even a complicated one, is fairly well understood, as applied e.g. in [29, 24]. However, if the object language is higher-order, the algorithm becomes much more

complicated [42]; the second-order nature of Idealised Accelerate avoids most of that difficulty, but it makes designing the AD algorithm already slightly less straightforward.

The second part, dynamic control flow, is also known to add difficulty for a reverse AD algorithm: in the absence of dynamic control flow, it is sometimes even possible to take a program using forward AD and fix its time complexity to match reverse AD using loop optimisations and generalisations of the distributivity law (see Section 2.2.4). The standard data structure for storing the primal values in reverse AD is a "tape" (see Section 2.1.3); if multiple executions of a program have the same path through the control flow graph, the tape memory and layout may be reused[50] (retaping, see Section 2.1.4). Without dynamic control flow, such a tape can *always* be reused; with dynamic control flow, this becomes more difficult. Where and how to store the primal values is an important and difficult question to answer when performing reverse AD on a language with dynamic control flow; our handling of conditionals gives a possible answer for the non-looping case, but loops are still challenging.

3.5 Expression transformation

Idealised Accelerate is a stratified language containing two sublanguages: the array sublanguage (called *arr* in Fig. 1) and the expression sublanguage (*exp* in Fig. 1). Our reverse AD algorithm for Idealised Accelerate is split into two parts handling these sublanguages separately, of which the array part was described in Section 3.4. The expression AD algorithm, which was used as a black box before, is what we will cover in this section.

We will assume the reader understands the array AD algorithm in Section 3.4. This is because the two algorithms are very similar in structure, and we wish to prevent a large amount of duplicate exposition. We will do this by describing only the structure of the expression AD algorithm and the major differences between the array and expression algorithms; while some details may end up underspecified, we believe that a motivated reader should be able to nevertheless construct the correct algorithm by extrapolation from the reasoning behind the array AD algorithm.

Just like the array AD algorithm, the expression AD algorithm is split in a primal phase and a dual phase that produce builders (this time for production *exp*); a gradient function could be produced from those builders in the same way as was described in Section 3.4.5. However, here our task is not to produce a single gradient function as output, but to conform to the interface as specified in Section 3.4.1 so that expression AD can be used in array AD. Therefore, we will wrap the primal builder in an expression function to produce the primal lambda, and the dual builder in an expression function to produce the dual lambda.

For both the primal phase and the dual phase, this divides the task of describing the algorithm up into two parts: how to produce the builder, and how to wrap the builder in an expression function with the right interface. Both Section 3.5.1 (for the primal phase) and Section 3.5.2 (for the dual phase) are divided in two in this way, first describing the creation of the builder, and then showing how to wrap the result in the correct expression function.

Labeling. The first step in the array AD algorithm was to label all *arr* nodes (except `Alet`) in the AST of the array function body with a unique label (Section 3.4.2). Here we do the same: we are given an expression function (of production *fun*), which is of the form `Lam x body`; we assume all *exp* nodes in the `body`, except `Let`, have a unique label, which we will again write with a subscript. For examples of labeled expressions, see later in this section.

[50] Unless some of the intermediate values are arrays with a different size each execution. However, even in this case, if the tape contains just the operations performed and only pointers to larger intermediate values, tape reuse can still occur and can also still be useful.

3.5.1 Primal

The primal phase of expression AD is a recursive traversal over the body of the `Lam` in the input expression function. It has the same input and output values as the primal traversal in array AD: given an expression (of production *exp* in Fig. 1) and a variable environment mapping variables to the label of their bound right-hand side, the recursive algorithm returns a builder for production *exp* and a to-store set containing expression variable names. For the expression primal, the purpose of the to-store set is to be able to handle conditionals correctly and to construct the temporaries tuple (t_{tmp}) when wrapping in the primal lambda.

Again we have a set of "regular"[51] expression primitives that follow the simple recipe, and a few that we will cover individually—for expressions, these are `Let`, `Cond` and `Index`. As a general note: where we used `a`-prefixed variables in the array primal, we will use `t`-prefixed variables in the expression primal. This choice is of course arbitrary, but a choice has to be made nevertheless.

For the regular primitives, the algorithm is recursively called on any *exp* arguments producing builders and to-store sets. In the array case there were expression arguments in which array variable references needed to be renamed; here there are no such embedded variable references. The builder for the current node is an incomplete let-binding that binds `t`n (where n is the label of the current node) to an expression that is the original primitive with the *exp* arguments replaced with `Evar` references to `t`m, where m is the label of the respective argument expression.[52] The output builder is then the composition of the argument builders and this incomplete let-binding, where the argument builders come first. The output to-store set is the union of the argument to-store sets with `t`n as an additional element.

For `Let`, the primal transform is fully analogous to `Alet` in Section 3.4.3.

For conditionals with `Cond`, the transformation is very similar to the array case, and is in fact simpler. Names are chosen to correspond with the description of the handling of `Acond` in Section 3.4.3. The first *exp* argument (the condition) is treated as a usual primitive argument, and is recursively transformed with its builder prepended to the result we will produce for the `Cond` itself. For the branches, we use the to-store sets s_1 and s_2 to create (expression) tuple types $\bar{t}^1$ and $\bar{t}^2$ that can store those variables, and define expressions e^k and f^k for $k \in \{1, 2\}$ analogously to the array case: e^k contains `Evar` references to the variables, and f^k is a tuple expression with the correct invocation of ZeroValue() in each position.[53] Analogously creating e_{then} and e_{else}, the output builder is like the one presented for the array case, but with `a`-prefixed variables changed to `t`-prefixed variables. Additionally, `Evar` is substituted for `Avar`, etc.

Finally, `Index` nodes are the only specialty in the expression primal transform. Suppose that the form of the current node is `Index`$_n$ `avar exp`$_m$. Then `exp` is recursively transformed as usual with its builder prepended to the builder for the `Index` itself, which is the following:

```
1  Let tn = Index avar tm
2  in Let tnexec = Const true
3  in _
```

The output to-store set is the union of the to-store set from `exp` and the set $\{\texttt{t}n, \texttt{t}n\texttt{exec}\}$. This `t`$n$`exec` variable becomes useful only in the presence of conditionals: since we explicitly defined ZeroValue(`bool`) = `false` on page 47, at the end of the expression, `t`n`exec` will be `true` precisely if the `Index` operation was indeed executed. Indeed, if the `Index` node is within the branch

[51] We allude to no particular special meaning of the word "regular", except that these primitives do not get special attention in this section.

[52] We do not provide a table like Table 1 for expressions, because its contents would follow directly from the description here and be analogous to the array algorithm.

[53] Contrary to the array case, we need no `Generate` here to create the shadow expression; just a zero value is sufficient.

of a `Cond`, its primal execution will also be (due to the construction of the builder for `Cond`), and importantly the binding for `t`n`exec` will be in the same branch. While for a single `Cond` its condition expression could also be used for the same purpose, this becomes harder with multiple layers of conditionals, where one would need to take the conjunction of the boolean conditions. Writing this additional `t`n`exec` variable solves the problem in a simpler way.

Wrapping. With this, we have described the algorithm to produce the primal builder for the input expression function. However, just producing a builder is not enough: we have to present the results from the primal phase in the form of the primal lambda as defined in Section 3.4.1. If the input expression function has type $t \to t'$, this primal lambda has type $t \to (t', t_{\text{tmp}})$.

As t_{tmp} we will choose the tuplified form of the to-store set produced from the primal transformation. The tuplified form of a set of variables is like the $\bar{t}^k$ we built for `Cond`.

For example, suppose that the label of the body of the `Lam` is 1 (and hence the function result is put in `t1` by the primal builder), and that the to-store set is $\{$`t1`, `t2`, `t3`$\}$. Then the expression to be substituted in the primal builder is the following:

```
Pair (Evar t1) (Pair (Pair (Pair Nil (Evar t1)) (Evar t2)) (Evar t3))
```

This expression is of type $(t_1, ((((), t_1), t_2), t_3))$, where t_i is the type of `t`i.

3.5.2 Dual

The dual phase of expression AD is again a recursive traversal over the body of the `Lam` in the input expression function, and as in the primal case, the input and output of the traversal are the same as for array AD. Given an expression, a variable environment and a contribution map (label $\Rightarrow$ list of expressions), the recursive algorithm returns a builder for *exp*, a to-store set (containing variables) and an updated contribution map. Like in the array dual phase, the to-store set is only used to ensure that conditionals can preserve validity of variable references in the contribution map.

The initial variable environment is empty, and the initial contribution map is the single-entry map $\{n \Rightarrow$ [`Fst (Evar adjarg)`]$\}$ where n is the label of the `Lam` body; the name "`adjarg`" stands for an arbitrary unused name that will be the name of the argument variable of the dual lambda. We take the `Fst` of the lambda argument because of the type of the dual lambda as specified in Section 3.4.1.

For the dual, we will describe `PrimApp`, `Let` and `Cond` separately, and group the rest of the primitives in the general recipe. Where the array dual used `d`-prefixed variables, the expression dual will use `j`-prefixed variables.[54]

The general recipe is the same as the four-step recipe in Section 3.4.4. First we look up the adjoint contributions to the current node in the incoming contribution map, which we add elementwise. Contrary to array AD, we do not have to treat certain summands in a special way; we can simply take the tuple-elementwise sum.

The contribution map additions for step 2 are mostly evident: `Const` is analogous to `Use` in Table 2; `Evar`, `Nil`, `Pair`, `Fst` and `Snd` are analogous to the corresponding `A`-prefixed version for arrays, replacing GenZero(x) with ZeroValue(*type of* x). A `Shape` node adds no contributions, and an `Index` node makes one contribution to its expression argument that is the variable containing its own adjoint.

[54] This is for lack of a better name. **A**rray variables, **d**uals and **t**emporaries are natural, but the "a" of "adjoint" is already taken. Thus we use ad**j**oint. It is not strictly necessary to use something else than "d"—the expression dual will never refer to array adjoint variables, and besides that, references are namespaced—but for clarity we separate them anyway.

After recursing on the expression arguments, threading the contribution map (including our additions) through, in step 4 the builder b is simply `Let` $\texttt{j}n$ `=` `@` `in` `_`, where `@` is replaced with the adjoint computed in the first step.

This leaves the dual for `PrimApp`, `Let` and `Cond`.

`PrimApp` works the same as the primitives covered above, but we want to emphasise some points nevertheless. The derivative of a `PrimApp` is the usual derivative function of the contained operation applied on the adjoint of the `PrimApp` node. Operations (production *op* in Fig. 1) with a fully discrete result, or a fully discrete argument, cannot propagate nonzero adjoints and therefore need to make no contributions at all.[55] As an example of a non-discrete operation, the contribution map additions for $\texttt{PrimApp}_n$ `Add` $\texttt{exp}_m$ are $\{m \Rightarrow [\texttt{Pair (Evar j}n\texttt{) (Evar j}n\texttt{)}]\}$, because $\frac{\partial}{\partial x}(x+y) = \frac{\partial}{\partial y}(x+y) = 1$. For `Mul`, we have the partial derivatives $\frac{\partial E}{\partial x} = \frac{\partial E}{\partial (x \cdot y)} \frac{\partial (x \cdot y)}{\partial x} = \frac{\partial E}{\partial (x \cdot y)} \cdot y$ and $\frac{\partial E}{\partial y} = \frac{\partial E}{\partial (x \cdot y)} \frac{\partial (x \cdot y)}{\partial y} = \frac{\partial E}{\partial (x \cdot y)} \cdot x$, so we get the following contribution map additions:

$$\{m \Rightarrow [\texttt{Pair (PrimApp Mul (Pair (Evar j}n\texttt{) (Snd (Evar t}m\texttt{))))} \\ \texttt{(PrimApp Mul (Pair (Evar j}n\texttt{) (Fst (Evar t}m\texttt{))))}]\}$$

Let-bindings with `Let` work exactly the same as `Alet` in the array dual, except that, of course, expression variables are used where the array dual used array variables.

For conditionals with `Cond`, the expression dual of `Cond` is to its expression primal as the array dual of `Acond` is to its array primal.

Wrapping. The text above describes how the expression dual algorithm produces the dual builder, as well as the to-store set and the contribution map for the whole expression. When constructing the primal lambda, we used the to-store set to determine t_{tmp} and its collecting expression; in the dual, as mentioned at the beginning of this section, we only used the to-store set to allow correct treatment of conditionals. For creating the dual lambda, we will use only the output contribution map from the dual algorithm.

The dual lambda from the interface to expression AD as defined in Section 3.4.1 is of type $(t', t_{\text{tmp}}) \to (t, t_{\text{idxadj}})$, if the original function was of type $t \to t'$. The structure of the expression function that we will create is as follows: restore the temporaries tuple into $\texttt{t}N$ variables; apply the dual builder to compute all adjoints; return a pair of the final adjoint and the *index adjoint collector*. This follows the structure shown in the example in Section 3.4.1.

The final adjoint is the elementwise sum of the contributions to the global function argument in the contribution map returned by the dual algorithm. Call this expression e_{argadj}, the "argument adjoint".

The index adjoint collector should be an expression of a to-be-determined type t_{idxadj} that contains, for each `Index` node in the function body, its computed adjoint, the target index, and a boolean indicating whether it was executed. If the `Index` node is of the form $\texttt{Index}_n$ `avar` $\texttt{exp}_m$, these values are available, respectively, in the variables $\texttt{j}n$, $\texttt{t}m$, and $\texttt{t}n\texttt{exec}$. Hence, if the i'th `Index` node has result type t_i and index shape type s_i, and there are k `Index` nodes, a possible choice for t_{idxadj} (as taken in the example in Section 3.4.1) is the following:

$$\big(\ldots\big(\big((), ((t_1, s_1), \texttt{bool})\big), ((t_2, s_2), \texttt{bool})\big)\ldots, ((t_k, s_k), \texttt{bool})\big)$$

The corresponding index adjoint collector expression would be:

[55] Recall that if the contribution list is empty when collecting the adjoint of a node, ZeroValue() is used.

```
Pair (...Pair (Pair Nil
                    (Pair (Pair (Evar jn1) (Evar tm1)) (Evar tn1exec)))
               (Pair (Pair (Evar jn2) (Evar tm2)) (Evar tn2exec))
      ...)
      (Pair (Pair (Evar jnk) (Evar tmk)) (Evar tnkexec))
```

where the k `Index` nodes are of the form $\texttt{Index}_{n_i}$ `avar exp`$_{m_i}$ for $i \in \{1, \ldots, k\}$. Let e_{idxadj} be this expression.

Now, the dual lambda is an expression function `Lam adjarg body`, where `body` is the builder composition $b_{\text{restore}} \circ b_{\text{d}}$ with `Pair` (e_{argadj}) (e_{idxadj}) substituted in the hole, where:

- b_{restore} is a composition of incomplete let-bindings that define, for all labels n stored in the temporaries tuple by the primal phase, `t`n as a `Fst`/`Snd` expression picking the correct value from `Snd (Evar adjarg)`. This follows the example in Section 3.4.1;
- b_{d} is the dual builder.

This defines the dual lambda, and thereby finishes our description of the AD algorithm.

4 Implementation

We implemented[56] the algorithm described in Section 3 in Haskell as an AST transformation in the Accelerate compiler. In Section 4.2 we give some implementation experiences and explain the major ways in which the implementation differs from the algorithm in Section 3, and in Section 4.3 we describe how the implementation has been tested for correctness. The subject of performance testing we delay until Section 5.

Because we stay at a relatively high level of abstraction, the only knowledge of the internals of the Accelerate compiler is the way it reifies types of the embedded language as values; this we discuss first in Section 4.1. We do assume knowledge about Haskell and GADTs, which are core topics in the discussion of an implementation of an algorithm in Haskell using type-indexed ASTs.

4.1 Reified types

In the Accelerate compiler, we often need to represent a type of the embedded language as a value in Haskell. One example place is in the `Const` constructor of the AST, which is defined as follows:

```
Const :: SingleType t -> t -> OpenExp env aenv t
```

The precise meaning of the `env`, `aenv` and `t` parameters of the `OpenExp` datatype encoding an expression AST, will be discussed later in Section 4.2.3.

This AST node, corresponding to the Idealised Accelerate constructor in Fig. 1 of the same name, contains the value itself (of type `t`) and a value-level representation of the *type* of the value. We need this information because, while Haskell is a polymorphic language, the Accelerate language itself is actually not. Therefore, for an expression `PrimApp Mul (Pair (Const 1) (Const 2))`, the compiler must know what type this expression has, which it cannot read just from the operations performed. For this purpose, `Const` (and `PrimApp`, and other AST constructors) contain *reified types* to represent the type of the expression. Some of the GADTs used for these reified types are shown below; note that in particular `SingleType` is very much abridged from the corresponding type in the compiler, called `ScalarType`, which supports a more extensive type hierarchy and spans multiple GADTs.

```
data SingleType t where
    TypeInt    :: SingleType Int
    TypeFloat  :: SingleType Float
    TypeBool   :: SingleType Bool

data Tup f t where
    TupUnit    :: Tup f ()
    TupSingle  :: f t -> Tup f t
    TupPair    :: Tup f t1 -> Tup f t2 -> Tup f (t1, t2)

data ArrayR arr where
    ArrayR     :: ShapeR sh -> Tup SingleType t -> ArrayR (Array sh t)

data ShapeR sh where
    ShapeRz    :: ShapeR ()
    ShapeRsnoc :: ShapeR sh -> ShapeR ((), Int)
```

`SingleType` models what Idealised Accelerate calls single-types (Section 3.1). `Tup` models tuples of a single-argument type constructor, for example `SingleType`. As an example, the embedded type $(\texttt{int}, \texttt{float})$ is represented as `TupPair (TupSingle TypeInt) (TupSingle TypeFloat)`. Finally, an array type is represented using `ArrayR`, which contains its shape type (the array's

[56] In the future, the aim is to merge (a later version of) the AD implementation created here into the Accelerate compiler proper. In the meantime, the code may be found in the `src/Data/Array/Accelerate/Trafo/AD` directory of: https://github.com/tomsmeding/accelerate/tree/e7a558ecf1e9d4baa0aa1fc7f09fc0b93c7fab55.

rank expressed in unary) and its element type. Note that `Array` (not `ArrayR`) can be seen just as a tag type here; in the compiler it has further meaning, but that meaning is not relevant for the AD pass.

Additionally, to index into a tuple, we define the following type that did not yet exist in the compiler:

```
data TupleIdx t' t where
    TIHere  :: TupleIdx t t
    TILeft  :: TupleIdx a t -> TupleIdx (a, b) t
    TIRight :: TupleIdx b t -> TupleIdx (a, b) t
```

As an example, `TILeft (TIRight TIHere)` has type `TupleIdx ((a, t), b) t`; the first argument is the "large" tuple type, and the second argument is the type selected from that tuple type.

The types defined in this section will be used below when we discuss some of the implementation details.

4.2 Implementation experiences

In this section, we describe how we approached writing the implementation (Section 4.2.1); we give an overview of the structure of the implementation (Section 4.2.2); we indicate some of the difficulties in translating the theoretical algorithm to something that works with the Accelerate internals (Section 4.2.3); and we indicate some patterns that empirically occur relatively often in our new code, as compared to the existing Accelerate code (Section 4.2.4).

4.2.1 Methodology

Work on a prototype implementation was started very early in the research process, immediately after an initial, high-level design of the algorithm was finished. This prototype implementation was quickly promoted to one for the Accelerate language itself, and incrementally expanded, covering larger and larger subsets of the language, until the language subset covered in Section 3 was achieved. Further expansion was desired, but due to time constraints unfortunately not achieved.

This incremental construction of the implementation is akin to a feature (build and iterate) of the Agile software development model, as opposed to the traditional waterfall method (design then build). We found that this approach proved quite advantageous, in that many problems in details of the algorithm, and underspecified portions, could be found and fixed soon.

Nevertheless, one downside of this approach is that some language features, which in isolation could be added to the implementation without much work, combine and interact in a way that demands a significant restructuring of the algorithm. This happened in particular with conditionals and array indexing.

- Without array indexing, there are no operations that can fail in the language, and hence implementing not a real conditional but instead a selection operator (evaluating both branches and choosing one based on the condition value) is a lot easier, and forgoes all the complexity in the handling of `Acond` and `Cond` in the algorithm.
- Without conditionals, all code is executed always, and hence we do not need to concern ourselves with determining whether an `Index` node was executed or not when computing its adjoint contributions.

Together, however, these two language features require the relatively complicated, separate handling of conditionals in the algorithm, as well as the extra `t`n`exec` variable in the dual of

`Index`; both of these play intricately with the exact definition of the ZeroValue() macro, which must produce `false` for the boolean type in order for `t`*n*`exec` to get the right value in the dual.

Absent powerful methods to prove, or otherwise attain confidence in, the correctness of an algorithm on paper within a reasonable amount of time, and since an actual implementation can be checked with tests, we do still believe that starting implementation of a new algorithm early on has a positive effect on productivity. This is despite downsides we encountered as described above in implementing an incomplete algorithm.

4.2.2 Overview

As already stated in Section 2.3.5, the implementation of our AD algorithm in the Accelerate compiler sits directly after sharing recovery (see Section 2.3.4) and before the actual optimisations in the compiler. The new compiler pass searches for the (new) `gradientE` or `gradientA` nodes, respectively replacing their contained subtree with the expression gradient[57] or array gradient transformation of the function under that node.

The transformation itself is implemented on a slightly expanded AST as compared to Accelerate's existing AST type, and includes labels on all nodes, new nodes for explicitly referencing the global function argument and free variables in the original function, and some auxiliary information needed in e.g. the `Index` node for the location of the `t`*n*`exec` variable. The translation between Accelerate's main AST and the special AD AST is straightforward, since the basic structure is the same.

The implementation of the transformation has the same three-part structure as the algorithm described in this thesis: labeling, the primal transform producing a builder, and the dual transform producing a builder.

4.2.3 Difficulties

The internal AST of the Accelerate compiler is not, as is evident from the name, Idealised Accelerate (Section 3.1). Were it exactly that language, an implementation of the algorithm in Section 3 would be straightforward, and this section would not exist. Because of the differences between these two languages, there is some additional creativity required in implementing our AD algorithm in the Accelerate compiler; in this section, we aim to shed light on some of the most important points.

Variables. Idealised Accelerate variables could have any type, including tuples (pairs), and includes projection functions `Fst`/`Snd`/`Afst`/`Asnd` to extract items from pairs. In the Accelerate AST, however, this is not the case. While expressions (and array programs) may of course have tuple types, *variables* are always single-typed (or, in the case of array programs, array-typed). Instead of including projection functions in the language, let-bindings always forcibly destructure their right-hand sides into the constituent single (non-tuple) components, binding each of them to a new variable.

We focus on the expression sublanguage here, because the array case is fully analogous.

In order to be able to *produce* code in this altered language, we introduce a second namespace of variable names in the AD algorithm. The *upper namespace* of tuple-typed variables contains what corresponds to the variables in Idealised Accelerate, while the *lower namespace* of single-typed variables is used to denote the actual variables in the produced expression (analogously, array

[57] The expression gradient function is computed using the expression AD algorithm (Section 3.5) and just combining the builders as in Section 3.4.5.

program). To be clear: the output of the AD algorithm will only contain variables names in the lower namespace, but during the algorithm we keep track of the upper namespace as well.

During the algorithm, then, we thread through a mapping from a name in the upper namespace to a tuple of names in the lower namespace. This is called the *binding map*. It is passed down into recursive calls in the primal and dual phases, and threaded through and returned upward upon return. Whenever the algorithm as written wants to refer to a variable in the upper namespace, what really gets produced in the output expression (in the builders and contributions) is a tuple of variable references for the corresponding names in the lower namespace, found with a lookup in the binding map.

Furthermore, whenever the algorithm wants to create a new variable in the upper namespace using a let-binding, new lower-namespace names are generated to fill the tuple type of the bound value and put on the left-hand side of the let-binding. A new binding is created in the binding map that maps the desired upper-namespace name to the created lower-namespace names.

To be able to *transform* code that is already *written* in this altered language, we have to be able to cope with automatically destructuring let-bindings in the input expression. For the primal phase this is straightforward, but the dual phase needs some adjustment. We do this in the dual phase by remembering, for each bound variable on a left-hand side, the type and label of the right-hand side that it binds a component of. Whenever we want to put an adjoint contribution for such a bound variable in the contribution map in the dual phase, we instead put a contribution to the label of said right-hand side in the contribution map, where we fill the other components with ZeroValue(). (In the array AD algorithm, these become GenZero() programs.)

With these two changes, we can consume and produce code written in the altered language with automatically destructuring let-bindings.

Type-indexed AST. While this is not technically a different language to perform AD on, it is still a difference from the algorithm-on-paper that requires some additional mechanics in the algorithm. We will build on the changes described above that introduced the binding map.

As already stated in Section 2.3.3, the internal AST in the Accelerate compiler is type-indexed. This means that the type of an expression[58] in the compiler is of the form `OpenExp env aenv t`, where:

- `env` is a type-level list containing the types of the (single-typed) variables that are in scope at the position where this expression sits in the AST. This list is indexed using De Bruijn indices. It is represented in practice using nested pairs: an empty list is `()`, an environment with one binding of type t is `((), t)`, etc.
- `aenv` is another type-level list containing the types of the (single-array-typed) array variables that are in scope for this expression. This is used for `Shape` and `Index` constructors in the AST.
- `t` is the type of the expression itself.

As stated in Section 4.2.2, the AD algorithm really operates on an AST separate from the main internal AST of the Accelerate compiler; this special AST has some additional type arguments, but for the purpose of this section, these are not important. Hence, we base our discussion on the main AST's type, which is `OpenExp env aenv t`.

Consider an example, using Idealised Accelerate here to illustrate the case because it is (intentionally) very close to the actual Accelerate AST. In the following expression, the multiplication at the core of the let-stack would have the type `OpenExp ((env, Float), Float) aenv Float`, for

[58] In this part we will again look only at expressions, since the changes to the array algorithm are analogous.

some array environment `aenv` and where `env` is the environment of the whole expression shown (possibly `()` if it is at the top level):

```
Let x = Const^float 1.0
in Let y = PrimApp Add (Pair (Evar x) (Evar x))
in PrimApp Mul (Pair (Evar x) (Evar y))
```

Note that we indicate the type of the `Const` node using an explicit annotation, because otherwise the type of the expression would be ambiguous.

In the AD algorithm in Section 3, we used simple strings as variable names and left it up to the implementer of the algorithm to ensure that these are always in scope and refer to a right-hand side of the correct type. Now that we *are* the implementer, due to this type-indexed AST, the Haskell compiler forces us to prove that the variable references in our produced program are indeed always in scope and of the correct type.

In order to do this, we introduce a new data structure, in addition to the binding map introduced above. This new data structure represents the environment of lower-namespace variable names currently in scope, and it is passed down in the algorithm, never up. This data structure can be defined as follows:[59]

```
data VarEnv env where
    Empty :: VarEnv ()
    Push  :: VarEnv env -> TypedInt SingleType t -> TupleIdx t' t -> TypedInt (Tup SingleType) t'
          -> VarEnv (env, t)

data TypedInt s t = TypedInt Int (s t)
```

There is a lot to unpack here. Firstly, recall `SingleType` and `Tup` from Section 4.1. The `TypedInt` type is used in this code snippet as both a lower-namespace variable and an upper-namespace variable. The actual implementation makes these types distinct on the type level to prevent namespace confusion, but we elide that here for simplicity.

The `VarEnv` type, then, contains a number of values for each binding in the environment (of single-typed variables). Of each such binding we store:

- The type and lower-namespace name of the bound variable in a `TypedInt SingleType t`.
- The (tuple) type and upper-namespace name of the let-bound right-hand side that the bound variable is a part of, in a `TypedInt (Tup SingleType) t'`; recall from above that let-bindings forcibly destructure their right-hand sides, so we give the original right-hand side a distinct upper-namespace name (an integer).
- Where the bound variable belongs in the bound tuple, in a `TupleIdx t' t`.

Consider, in a modification of Idealised Accelerate where let-bindings destructure their right-hand side, the example expression `Let (x1, x2) = expr in body`; assume that `expr` has type `OpenExp env aenv (Float, Int)`. Note that this means that the environment type of the whole `Let`-expression is also `env` (since Accelerate does not have recursive let-bindings, so a let-expression and its right-hand side have the same environment). The type of `body` is then `OpenExp ((env, Float), Int) aenv t` for some `t`. Assume that the upper-namespace name of `expr` is 100, and that the lower-namespace names for `x1` and `x2` are 1 and 2, respectively. The `VarEnv` belonging to the context for `body` would then be:

[59] The actual definition in the implementation (called `LabVal`, instantiated on `PartLabel`) differs slightly, but for the purposes of this text it is equivalent.

```
Push (Push varenv (TypedInt 1 TypeFloat)
                  (TiLeft TiHere)
                  (TypedInt 100 (TupPair (TupSingle TypeFloat) (TupSingle TypeInt))))
     (TypedInt 2 TypeInt)
     (TiRight TiHere)
     (TypedInt 100 (TupPair (TupSingle TypeFloat) (TupSingle TypeInt)))
```

This pushes two entries onto an existing `VarEnv`, named `varenv`, corresponding to the environment of the whole `Let`-expression. Note that e.g. `TiRight TiHere` has type `TupleIdx (a, b) b`.

In the way we describe here using `VarEnv`, we can keep track of the variables that are in scope in the lower namespace, including the upper-namespace name that they are part of and including a `TupleIdx` for tracking where in the upper-namespace value the lower-namespace value should be put for adjoint contributions.

A criticism one can have about the `VarEnv` data type defined above is that looking up a variable in this environment has linear complexity in the size of the environment. This is true, and is something that the entire Accelerate compiler currently suffers from, since these linear environments (in different forms) occur elsewhere in the compiler, and in particular all typed De Bruijn indices are in unary representation. A possible fix for this performance problem is to use a data structure with logarithmic lookup time, for example the one defined in [54]. Worth noting is that while using linear environments is a possible performance problem in the compiler, the compiled programs do not suffer from this inefficiency: the actual compiled code that gets run on the GPU, or on multicore CPU, is optimised via LLVM.

4.2.4 Patterns

The type-indexed AST has consequences for the types of any code in the Accelerate compiler that deals with AST-related values. Since our algorithm is an AST transformation, there is in fact little code that is not affected by this.

However, despite this effect, we find that we do not need any significant type-system extensions over what the Accelerate compiler already uses elsewhere. In particular, we do need support for GADTs to work with a GADT-based AST, and we also use higher-rank types fairly heavily. In this section we will discuss in a bit more detail our use of higher-rank types and existentials, as well as our use of runtime type-checking.

Builders. The concept of a *builder* was introduced in Section 3.4.3, and refers to an AST fragment with one hole. In the implementation, we represent a builder (the expression version here) using a higher-rank type:

```
data EBuilder env aenv =
    forall env'.
        EBuilder (EContext env')
                 (forall res. OpenExp env' aenv res -> OpenExp env aenv res)
```

In addition to the builder function itself, which is the second argument of `EBuilder`, this data type also contains the context in the builder hole (containing the variable environment (`VarEnv`) and binding map from Section 4.2.3).

This definition contains two occurrences of the keyword `forall`, with two different meanings.

- The first is enabled using either the GADTs or the ExistentialQuantification language extension in GHC, and denotes an existentially quantified type variable, or an *existential* for short. The producer of an `EBuilder` may choose the `env'` type at will, and it will not appear in the externally-visible type of the `EBuilder`. A consumer of an `EBuilder` has no information about the `env'` type except via the values contained within the `EBuilder`; in

this case, all necessary information is in the variable environment in the `EContext`, which contains in essence a reification of the `env'` type by means of the `VarEnv`.

- The second occurrence of `forall`, in the second argument of `EBuilder`, is allowed using the RankNTypes[60] extension and denotes a *universally* quantified type variable. This makes the second argument of `EBuilder` a *rank-2 type*, hence the name of the language extension. This function guarantees that given an expression with the particular `env'` and `aenv` type parameters as specified by the `EBuilder`, but for *any* result type `res`, the output will be an expression with the same result type. The function has no knowledge about this `res` type variable except for information contained (in reified types) in the `OpenExp` value. In practice, even this information will not be used by a builder.

Higher-rank data types, that is the second usage of `forall`, are a natural extension to the type system that bring polymorphic functions to (almost) equal footing with monomorphic functions, by allowing us to put them in data structures.

Existentials, and GADTs which allow a generalisation of this behaviour, bring a limited amount of dependently-typed programming to Haskell. In fact, the `EBuilder` type given above can also be written in GADT syntax, which may be more familiar for dependently-typed programmers:

```
data EBuilder env aenv where
    EBuilder :: EContext env'
             -> (forall res. OpenExp env' aenv res -> OpenExp env aenv res)
             -> EBuilder env aenv
```

Data types with existentials have a duality via code in continuation-passing style (CPS). Indeed, a function that returns a builder:

```
foo :: Something -> EBuilder env aenv
```

can also be written without existentials by taking a continuation:

```
foo' :: Something
     -> (forall env'.
             EContext env'
          -> (forall res. OpenExp env' aenv res -> OpenExp env aenv res)
          -> a)
     -> a
```

This function `foo'` takes as its second argument a function, that in turn takes the fields of `EBuilder` as its arguments. Note that the `forall` in `EBuilder` that indicated an existential type has morphed into a normal, universal `forall` in the CPS version. We have the following equivalences:

```
foo something = foo' something EBuilder
foo' something k = case foo something of EBuilder c f -> k c f
```

To avoid many explicit continuations in the code, we generally opt for the version using existentials. This style is in particular less unwieldy when in a long monadic computation, which occur fairly often in our implementation to be able to generate unique ID's in various places.

Interface to expression AD. A different use of existentials (as described above) is in the interface to expression AD, the algorithm version of which was defined in Section 3.4.1. Expression AD returns two expression functions as well as some information about the `Index` nodes in the expression. This data contains two existential types: t_{tmp} and t_{idxadj}, respectively indicating the type of the tuple containing the temporaries stored in the primal lambda, and the type of the tuple containing information about the adjoints and target indices of the `Index` nodes.

[60] Or, technically, the Rank2Types extension, which is subsumed by the more often used RankNTypes.

From the perspective of array AD, these types are new: as we already observed in Section 3.4.7, these types did not already occur in the program that was given as input to our AD transformation. Hence it is unsurprising that in the implementation, these types are indeed encapsulated in a data structure using existentials. However, that data structure does contain reified versions for those types, so that array AD can work with these types even if it does not inspect them explicitly: the Accelerate AST contains reified types in many nodes, and in particular array primitives generally have their result type embedded as a reified type. Since values of type t_{tmp} and t_{idxadj} are put in arrays, array AD needs access to the reified versions of these types in order to construct the output AST.

The rest of the compiler. The reason why we highlight our usage of existential types, which may be natural for the seasoned Haskell or dependently-typed programmer, is that the rest of the existing Accelerate compiler seldomly uses this construction. GADTs are used extensively, but explicit abstraction of certain types without further usage of a GADT may be novel in the compiler. A possible reason for this is that the compiler passes that are already implemented are generally structure-oriented, preserving (components of) the structure and types of the original program, whereas our AD pass changes the program structure significantly and introduces new types.

Runtime type checking. As announced in Section 2.3.6, implementing our reverse AD algorithm in Accelerate required some runtime type checking. What we mean with this term is in essence usage of the following function:

```
1 matchSingleType :: SingleType t -> SingleType t' -> Maybe (t :~: t')
2 matchSingleType TypeInt   TypeInt   = Just Refl
3 matchSingleType TypeFloat TypeFloat = Just Refl
4 matchSingleType TypeBool  TypeBool  = Just Refl
5 matchSingleType _         _         = Nothing
6
7 data t :~: t' where  -- from the standard module Data.Type.Equality
8     Refl :: t :~: t
```

Here, the `Refl` constructor of the `:~:` type witnesses equality of types. Similar functions to `matchSingleType` exist also for `Tup`, `ArrayR`, etc. from Section 4.1.

Using these functions is necessary whenever GHC requires a proof that two values have the same type, but we did not model the problem in the type system in a detailed enough manner for validity to be evident at compile time. The most important case where this happens is when looking up a lower-namespace variable in the `VarEnv` from Section 4.2.3. The variable we are looking up is represented by what boils down to a `TypedInt`; looking that up in the variable environment entails looking for the integer and then checking that the types match. GHC cannot deduce the equality of types from the code itself, because there is no link between the type in the `TypedInt` and the corresponding element in the type-level list that is the `env` type parameter of the `VarEnv`.

In the introduction to this thesis and in Section 2.3.3, we stated that because the internal AST of the Accelerate compiler is type-indexed, the implementation of our algorithm must, at each step, be at least type-correct, if not necessarily semantically correct. It must be noted that the necessity of runtime type checks does not invalidate this guarantee: it merely means that GHC requires us to connect some of the loose ends of the correctness proof by supplying evidence at runtime. In essence, either our algorithm implementation performs no type-incorrect program transformations, or it will lead to an early crash at runtime because a caller of `matchSingleType` (or similar functions) receives `Nothing` instead of an equality proof. At no time can the algorithm

finish, without crashing, and produce an AST that is type-incorrect.[61]

Nevertheless, the situation of requiring runtime type checks is not ideal, if only because it is essentially redundant computation, assuming that the algorithm is indeed correct. However, ensuring that these checks are not necessary may require a significant amount of work to prove type-correctness of our algorithm in the Haskell type system, and we have not attempted to do this.

4.3 Testing

In Section 5, we will discuss two larger Accelerate programs that use our reverse AD implementation. These programs are useful not only for performance testing, but also correctness testing, since we have input data and accompanying correct output data for these programs (see Section 5 for more details). We verified that for the input files for which our implementation succeeds within the resource constraints defined for the experiments, its outputs do not differ significantly (relative to the magnitude of the input values) from the control values computed by the reference C++ implementation of the benchmark tasks. The variation that does occur is mainly due to the fact that we compute with single-precision floating-point numbers, whereas the control values are computed using double-precision numbers. In the development of our implementation, we have found implementation errors by observing that the output of the benchmark programs did differ significantly from the control values; empirically, we can thus say that the current non-significant difference has some meaning.

However, besides the benchmark programs, we wish to also test our implementation in a somewhat more structured way. Because we do this by writing a number of unit test programs, we need to check whether our reverse AD algorithm produces the correct derivative for these test programs. Manually writing the correct derivative program and checking that it matches the implementation's output is tedious and very error-prone; fortunately, however, the task of computing the gradient of a function has a very fortunate property: there are multiple algorithms for it.

These algorithms range from slow and simple to fast and complicated. Reverse-mode automatic differentiation, as we implement in this thesis, is on the right-hand side of that scale: it has low time complexity, but is quite complicated to design and implement. Aside from reverse AD, there are two other methods that are significantly simpler to implement, but also significantly slower in terms of runtime. Assume that the program in question has n input values and 1 output value.

- One such method is forward AD. As we saw in Section 2.1, reverse AD computes a row of the Jacobian while forward AD computes a column of the Jacobian. If we (somehow) cannot use reverse AD, then certainly another way to compute the (one-row) Jacobian of our program is to use forward AD on every individual one-element column, and stitch the results together. This corresponds to running the program with forward AD once for each of its input values, each time computing the tangent of the output value with respect to this input.

  This runs the program n times, whereas reverse AD would run the program only once.[62]

[61] Any claim of this audacity requires a footnote stating its limitations. Firstly we define "finish" to include `deepseq`'ing the resulting AST. We assume that any of the existing uses of `unsafeCoerce` and `unsafePerformIO` in the Accelerate compiler do not introduce type-unsafety, and we assume the same for any of the libraries used in the compiler. Furthermore, of course, we assume no miscompilation by GHC and a processor that conforms to its specification, but those two assumptions are unlikely to be false in practice.
Our algorithm implementation does not introduce new uses of `unsafeCoerce` or `unsafePerformIO` into the compiler, apart from those in libraries that we use.

[62] Both forward and reverse AD add a different constant factor to the runtime that we ignore here, but for larger inputs, the effect of running the program once for each input element quickly starts to dominate.

- Another method is finite differencing. Recall the analytical definition of the derivative of a function $f : \mathbb{R} \to \mathbb{R}$:

$$\frac{\partial f}{\partial x}(a) = \lim_{h \to 0} \frac{f(a+h) - f(a)}{h}$$

  Surely, then, if we just pick a small h, say 10^{-5}, the expression $\frac{f(a+10^{-5})-f(a)}{10^{-5}}$ will be an approximation of $\frac{\partial f}{\partial x}(a)$?

  This works to an extent: for small enough h, the given fraction will indeed approach the derivative mathematically (and for many functions it will also do so reasonably quickly). However, a problem in practice is that computers do not work with infinite-precision reals; instead, they usually work with floating-point numbers, which exhibit *catastrophic cancellation* (loss of precision) when subtracting two numbers that are very close together. Because $f(a+h)$ and $f(a)$ are very close together by design, the precision of the derivative approximation given by finite differencing quickly degrades as h gets smaller.

  If h is taken large enough that the number of significant digits remains sufficient after subtraction but small enough that the difference quotient is already near the actual derivative, then finite differencing can be used to give an approximation of the gradient of the function in $O(n)$ function evaluations. We say $O(n)$ here and not n because at least one extra evaluation is necessary for $f(a)$, and if symmetric differencing (see below) is used, there will in fact be $2n$ invocations.

In short, this means that we have two separate alternative algorithms that should compute or approximate the same result as the one computed by reverse AD, if it is correct. By implementing both these alternative algorithms, we can test our implementation on arbitrary test programs without fear of having incorrect validation data for the test programs: if three completely distinct implementations (reverse AD, forward AD and finite differencing) all give (approximately) the same result, it is unlikely that all three are wrong.

We want to note that our forward AD and finite differencing implementations are written as Haskell-level libraries for Accelerate; no modification of the Accelerate compiler was necessary for these implementations. This is in stark contrast to the reverse AD implementation that we described in Section 4.2.

4.3.1 Unit tests

The unit test programs are hand-written to (collectively) include all the interesting constructs in the input language.[63] The general approach is that we use each of the unit test programs as a property test, checking that for all inputs of a particular (refined) type, the program produced by reverse AD gives the same gradient as forward AD and finite differencing. The programs themselves are given in the third column of Appendix A; the input names and their refined types are in the second column. All programs deal only with integers (`Int`) and single-precision IEEE-754 floating-point numbers (`Float`).

The gradient is in all cases computed with respect to the array arguments listed in the second column of the table. Any non-array input variables are considered constants from the perspective of differentiation.

For more information about the exact syntax of actual Accelerate programs, we refer to the Accelerate documentation [12].

[63] We originally intended to algorithmically generate arbitrary programs to test with our AD transformation, but this proved harder than anticipated; time constraints then forced us to accept a collection of hand-written programs instead.

The notation in the second column is as follows. After the ∀-sign, there are one or more variable names, written in `typewriter style`, with their refined type as a superscript. Each of these variables is considered in-scope in the program in the third column.

For the refined types, we use the following notation.

- V: a vector (single-dimensional array) of `Float` values of length 0 to 20, where the values are in the interval $[-10, 10]$.
- $Float$: a single `Float` value in the interval $[-10, 10]$.
- $[a..b]$: a single `Int` value from the set $\{a, a+1, \ldots, b\}$.
- $T|_{prop}$ for some refined type T: elements of T that satisfy $prop$, when substituted for v in $prop$.
- $\sum v$: the sum of the values in the vector v.
- $\#v$: the length of the vector v.
- even(x): checks that the integer x is even.
- allNonZero(v): checks that all elements x in the vector v satisfy the property $|x| > 0.1$.
- uniqueMax(v): checks that no more than one element in the vector v satisfies the property $x \geq m - 0.5$, where m is the maximum value of the vector v.
- allNiceRound(v): checks that for all elements x in the vector v, x is at most 0.4 separated from a whole number. The intent is permuting an element in finite differencing does not change the result of rounding the element to an integer.

Construction of unit tests. The first part of the list of unit tests consists of programs that are meant to test one particular language primitive; the last of these is `backpermute_2`. The programs whose names start with `aindex_` are meant to stress the algorithm handling array indexing. Following those are a number of tests that either fill an edge case (e.g. `a_ignore_argument`), are a regression test from an earlier version (`logsumexp`) or stress the tuple-handling machinery (`tuple`). The `neural` test is a simplistic implementation of a three-layer dense neural network, with weights sampled via some arbitrary formulas from the input vectors. `acond_cond1` uses both `acond` and `cond` to test that conditionals really execute only the branch that would be taken in the original program. Finally, on the last page of Appendix A are a few expression-centric tests, that mainly test condition handling.

Some of the test names have the suffix `_friendly`. These tests have the same code as the corresponding test without `_friendly`, but have additional refinements on the input types in order to avoid discontinuities; this allows finite differencing to succeed on these tests.

4.3.2 Methodology

As stated above, each of the programs in Appendix A is differentiated with respect to the listed array-typed inputs using our reverse AD algorithm, a simple forward AD implementation, and finite differencing. We use the Accelerate interpreter (as opposed to one of the native-code backends) to run our unit tests, because the interpreter includes bounds checks, allowing us to check that array indexing operations are always in-bounds. The gradient from reverse AD must always be approximately equal to the gradient from forward AD; if finite differencing produces a result approximately equal to forward AD, then that result must also be approximately equal to the gradient from reverse AD.

The parts of this process that are yet underspecified are precisely what algorithm is used for finite differencing, and what our definition of "approximately equal" is.

Finite differencing. To estimate a gradient using finite differencing, we estimate the partial derivative of the program with respect to every individual input element that is contained in an array-typed input. Let $f(x)$ be the output value of the program when one particular input element is replaced with x. Then we estimate the partial derivative, $\frac{\partial f(x)}{\partial x}(a)$, for this particular input element by computing the *symmetric difference* formula for every value h in the set $\{2^{-i} \mid 4 \leq i \leq 7\} = \{0.0625, 0.03125, 0.015625, 0.0078125\}$:

$$\text{symdiff}(f, a, h) := \frac{f(a+h) - f(a-h)}{2h}$$

Using the symmetric difference instead of the one-sided difference offers a slight defence against systematic bias in the estimate of the partial derivative.

The four computed values are then used to estimate the value at the hypothetical $h = 0$ using *ordinary least-squares regression*. This means that the (unique) reals u, v are found such that the following value is minimised:

$$\sum_{i=4}^{7} \left(u \cdot 2^{-i} + v - \text{symdiff}(f, a, 2^{-i})\right)^2$$

The estimate for the concerned partial derivative is then the value v.

The values of h used in the above process may appear to be quite large. However, it must be noted that the test programs use single-precision floating-point numbers; empirically, using smaller values for h resulted in significant loss of precision in some cases.

Error tolerance. A gradient is a tuple of arrays, each of which records the gradient with respect to a particular input array. Two tuples of arrays are *approximately equal* if all corresponding constituent arrays are approximately equal. Two arrays are approximately equal if they have the same size, and for all pairs of a floating-point number x in the first array and y in the second array, either the absolute difference $|x - y|$ is less than 0.1, or the relative difference $\frac{|x-y|}{\max\{|x|,|y|\}}$ is less than 0.1. We use this adaptive comparison method to allow for both some *additive* and some *multiplicative* error in the estimates (particularly for finite differencing).

4.3.3 Results

Forward AD agrees with reverse AD for all test programs: their output gradients are always approximately equal in the sense of Section 4.3.2. However, for finite differencing the story is unfortunately not so straightforward.

At the beginning of Section 4.3 we explained how finite differencing has precision issues if we take h to be too small. However, in addition to precision issues, the non-trivial magnitude of h leads to another property of finite differencing that is not shared by both methods of AD employed in this section: it is sensitive to discontinuities of the program being differentiated and its derivative.

For example, consider a program that takes only one input (i.e. it models a function $\mathbb{R} \to \mathbb{R}$) and is given by the function $f(x) = \max\{0, x\}$. This is also known as the rectified linear unit, or ReLU, in machine learning. While f is continuous, its derivative is discontinuous at 0.

Suppose we require the derivative of f at input value 0.001. Both forward and reverse AD will return exactly 1 in this case, irrespective of whether f is implemented using a primitive maximum-value function or using a conditional; even in the second case, 0.001 is firmly on one

side of the conditional, and AD will give the right answer. However, using symmetric finite differencing, we may compute $\text{symdiff}(f, 0.001, h)$ for $h \in \{2^{-i} \mid 4 \leq i \leq 7\}$, and obtain the following values:

h	2^{-4}	2^{-5}	2^{-6}	2^{-7}
$\text{symdiff}(f, 0.001, h)$	0.508	0.516	0.532	0.564

While these values are certainly moving in the right direction (towards 1), an estimate of the derivative at 0.001 based on these values will have a significant error. This is of course because the $f(0.001 - h)$ sample lies left of 0 for these values of h, while $f(0.001 + h)$ lies right of 0. The error from finite differencing can be even stronger if f itself is discontinuous near the point for which the derivative is required.

How does this work out on our test programs? Because not all our programs are continuous over their full domain, finite differencing fails to give an accurate result for some test programs. The ones for which finite differencing does not agree with forward AD are marked with a red star (*) before the name. On all other programs, finite differencing agrees (meaning that its result is "approximately equal" in the terminology of Section 4.3.2) with both forward AD and reverse AD.

All programs for which finite differencing does not agree with the other algorithms have discontinuities, either in the function they model or its derivative. These discontinuities generally result from deliberate "weirdness" introduced in the program by way of conditional array indexing operations; the intention of these programs is to test the reverse AD implementation, after all, and the interaction between conditionals and array indexing has proven to be the most difficult part of the algorithm. For many of those discontinuous programs, changing the scalar computations so that all the piecewise fragments align and form a continuous (and everywhere differentiable) function would be nontrivial.

For some programs, e.g. `cond_2`, the scalar expressions are designed so that at points where the conditional switches branches, the modelled function and its derivative are both continuous. For other programs, where conditional switching points can easily be removed from the input domain by refining the input type, we added a `_friendly` variant of the test program that includes this refinement (and has no changes in the program itself). For the rest of the programs, we accept the loss of finite differencing in checking correctness of our reverse AD algorithm.

5 Experimental results

In the introduction to this thesis, we promised that our implementation of the reverse AD algorithm not only works, but also produces efficient code. Correctness testing we have discussed in Section 4.3, so in this section we continue with performance testing. All testing will be on a many-core CPU (see Section 5.2.1); tests on a GPU platform are left to future work.[64]

Šrajer et al. published a suite of benchmark tasks (ADBench) for AD implementations and compared a number of existing AD implementations in various programming languages and frameworks [51]. Since the publication of that paper, one additional benchmark task has been added to the suite to make a total of four tasks. We implemented two of the original tasks using our reverse AD implementation: Gaussian model mixture fitting (GMM) and bundle adjustment (BA). The third task (hand tracking, HT) requires that the AD implementation support sparse Jacobians, and the fourth task[65] (diagonal long-short-term memory, D-LSTM) requires (bounded) looping for an effective implementation. Unfortunately, neither of these features are present in our reverse AD implementation; hence, we compare just based on the first two tasks.

In the following sections, we define the two tasks that we implemented, namely GMM and BA (Section 5.1), we describe the benchmarking environment and methodology (Section 5.2), and finally we show and analyse the benchmark results (Section 5.3).

5.1 Benchmarking tasks

As stated above, we implemented two benchmark tasks from the ADBench suite: GMM and BA. For semantic context of those tasks, i.e. why these tasks are meaningful, we refer to the article that originally defined them [51]. Here we assume the tasks given and discuss just their implementation in our extended version of Accelerate.

Some of the details of what the benchmark tasks should compute, precisely, are left abstract in the cited article. Therefore, we referred to some of the existing implementations in the ADBench repository[66] to determine the exact task definitions. In Appendix B we give our formulation of the task definitions, which we discuss here. The Accelerate source code of our task implementations is given in Appendix C;[67] we note that reading the implementations may unfortunately require knowledge of Haskell as well as familiarity with Accelerate.

GMM. The first task is a simple gradient task: in Appendix B.1 a function L is defined that computes a single real value based on its four arguments, $\boldsymbol{\alpha}, \boldsymbol{M}, \boldsymbol{Q}, \boldsymbol{L}$, and some additional values: an array $\boldsymbol{X}$ and two scalar inputs γ, m. The parameters N, D, K are size parameters that characterise the size of the input. The task is to compute the gradient of L, with respect to $\boldsymbol{\alpha}, \boldsymbol{M}, \boldsymbol{Q}, \boldsymbol{L}$, for a particular input data set specifying all mentioned parameters.

Highlighted in the definition of L are scalar values that are independent on the four arguments of L, and that we can therefore precompute outside of the gradient computation. This is fortunate: while the log MultiGamma function can be written out using the more commonly available `lgamma`

[64] The Accelerate GPU backend currently has an issue preventing effective testing on very large Accelerate programs, which our AD algorithm produces. Furthermore, the ADBench benchmark suite currently focuses solely on CPU. These two difficulties combined make us, too, focus only on CPU for this work.

[65] D-LSTM is a simplification of a real LSTM neural network; one of the important changes is that the weight matrix is a diagonal matrix. Our assessment of the task is purely based on scanning existing implementations, because documentation is unfortunately scarce. See the ADBench repository for more information: https://github.com/microsoft/ADBench/issues/143.

[66] https://github.com/microsoft/ADBench

[67] The actual programs used for benchmarking, including modifications to the ADBench framework to include our implementations, can be found here:
https://github.com/tomsmeding/ADBench/tree/157260330293a46068593357cebc0f71f203750b

function, `lgamma` is nevertheless currently unavailable in Accelerate. Therefore, we compute the highlighted portions of L outside Accelerate, and insert their results as constants in the Accelerate program.

Note that we overload exp to map over a vector argument, both in L and in the definition of logsumexp.

As for the implementation of the task, this was mostly a straightforward translation of the specification into an Accelerate function that simply computes the objective, i.e. implements the function L, and then uses `gradientA` to obtain its gradient. This is true except for one part: the computation of the matrix-vector product of the matrix Q and the vector $\boldsymbol{x} - \boldsymbol{\mu}$ in the second term of L. Firstly, we do not actually compute the matrix Q, but instead define ΔQ to be the diagonal of Q and compute $\Delta Q \cdot (\boldsymbol{x} - \boldsymbol{\mu}) + (Q - \Delta Q) \cdot (\boldsymbol{x} - \boldsymbol{\mu})$. The first of these multiplications is simply an elementwise vector product; for the second we perform a naive cubic matrix product.

For this matrix product we do fully represent $Q - \Delta Q$ in the program,[68] and the construction of this matrix (the lower-triangular part of Q) can be done in multiple ways. The simplest way to compute $Q - \Delta Q$ is using a `generate` that checks for each cell whether it is in the lower triangular part; if so, it picks the right value from $\boldsymbol{L}$, otherwise it places a zero. This definition involves array indexing (into $\boldsymbol{L}$), and while our reverse AD implementation can differentiate this, we found that it is not the most efficient option. Indeed, for an earlier version of the reverse AD algorithm that could not yet differentiate array indexing, we wrote a somewhat more elaborate way to construct $Q - \Delta Q$ that worked out in practice to perform better under AD than the version *with* array indexing. It is this version that we use in our implementation of the benchmark program, for use in benchmarking.

This alternative method to compute $Q - \Delta Q$ uses the following trick: the use of array indexing in the natural implementation is relatively tame, in that it could be implemented without array indexing using `backpermute` if only we could decide to return a certain default value (i.e. zero) for some cells, instead of being forced to always pick a value from the backing array ($\boldsymbol{L}$). However, we can simulate this by first extending $\boldsymbol{L}$ with an additional zero, and then backpermuting from this extended array. In computing this zero-extension, we make use of the fact that from the input file, we really have $\boldsymbol{Q}$ and $\boldsymbol{L}$ in a single $K \times (D + \frac{D(D-1)}{2})$ array, and that $K, D > 0$. First backpermute the right half of that input array (the $\boldsymbol{L}$ part) plus one element from $\boldsymbol{Q}$ (using the fact that $D > 0$), then elementwise-multiply the result with an array containing ones except at the first element, where it contains a zero.

For the full source code of our GMM implementation as it runs in the benchmarks, see Appendix C.1. Note the usage of `gradientA` to compute the final gradient.

BA. The second task is different from the first in that we are now required to produce a sparse Jacobian of a function with more than one output. However, the objective function being differentiated has many internal redundancies, due to which we can take a shortcut in computing the sparse Jacobian to ensure that we only really need an AD implementation that can compute a gradient. This shortcut is documented in [51] and also used in the other task implementations in the ADBench suite.

First, however, we discuss the objective function given in Appendix B.2. The values N, M, P are size parameters. The input data, which contains only one camera, one point, one weight and one feature pair, is enlarged to produce the arrays $\boldsymbol{C}, \boldsymbol{X}, \boldsymbol{w}, \boldsymbol{F}$. The *obs* array contains pairs of an

[68] We compute the whole matrix, including zeros, in the objective program; this does not necessarily mean that at runtime, after fusion optimisations, the whole matrix is actually *materialised* in memory. Whether that happens depends on how we use the matrix, and what AD does with the program.

index into $\boldsymbol{C}$ and an index into $\boldsymbol{X}$; we will only access $\boldsymbol{C}$ and $\boldsymbol{X}$ through the *obs* indirection.[69]

We define the functions 'reprojerr' and 'werr' using a number of helper functions; the dimensionality of their arguments is indicated using an overset number 1, 2 or 3. The objective function (which here computes an *array* of values, not a single value) is the concatenation of 'reproj_error' and 'w_error', where reproj_error is first flattened to an array of $2p$ values, organised in p pairs of 2 values.

This objective function (with $2p+p$ output values) is to be differentiated with respect to the input arrays $\boldsymbol{C}, \boldsymbol{X}, \boldsymbol{w}$, where $\boldsymbol{C}$ and $\boldsymbol{X}$ are first flattened in row-major order, the same way in which we flattened reproj_error. Concatenating the (flattened) inputs gives an array with $11n + 3m + p$ values, meaning that the to-be-computed Jacobian (called J in Appendix B.2) is a matrix of size $(2p + p) \times (11n + 3m + p)$.

Figure 4 shows the nonzero entries in a possible Jacobian for BA; note that we do not use the partial redundancy due to $\boldsymbol{x}_0$ at all. The non-zero values in a row in the top two-thirds of the Jacobian are the gradient of one of the two output values of reprojerr() with respect to a particular camera, point and weight as determined by *obs*; the single non-zero value in a row in the bottom one-third of the Jacobian is the derivative (gradient) of werr() evaluated on one of the weights. If we compute just these gradients, then placing them in the right positions in the full Jacobian is easy. This trick, where we compute just the blocks themselves instead of computing the full Jacobian without regard for sparsity, is the shortcut that we mentioned above; it is applied in the existing implementations in the ADBench suite, as well as ours.

Nevertheless, BA is a different kind of problem than GMM: we need many gradients with respect to a bounded number of values (namely, 15 and 1 for the reprojection errors and the weight errors, respectively) instead of one gradient with respect to a large array of values. Therefore, in our implementation of the BA task, we use our expression-level reverse AD primitive to compute the gradient of an expression function with respect to a tuple of input values. The resulting expression function can then run in parallel on all places in the Jacobian where it is required. This stands in contrast to the use of `gradientA` in GMM: since `gradientA` is an array-level primitive, the second-order nature of Accelerate prevents us from running many array-level gradients in parallel.[70]

For the full source code of our BA implementation as it runs in the benchmarks, see Appendix C.2. Note the usage of `gradientE` to compute an expression-level gradient. Note further that our BA implementation uses some metaprogramming (from the perspective of Accelerate) in Haskell to define `vecadd`, `vecsub`, `dot` and `scale` for both `Pt2D` and `Pt3D`.

5.2 Methodology

For both the GMM and BA tasks, the ADBench suite contains a number of input files. For GMM this is one input file for each triple (N, D, K) in the following Cartesian product:[71]

$$\{10^3, 10^4\} \times \{2, 10, 20, 32, 64, 128\} \times \{5, 10, 25, 50, 100, 200\}$$

The values in the input arrays are randomly generated.

[69] It is tempting to simplify and optimise one's implementation by using the artificiality of the input formulation, to the point where the description in [51] is ambiguous how much an implementer may assume, precisely. Looking at the other implementations, however, makes clear that the function to be differentiated cannot assume any redundancies in the five input arrays, including *obs*.

[70] Actually, some limited parallelism is introduced by Accelerate's runtime scheduler: on multicore CPU, a small number of array-level primitives may run in parallel on separate cores if they do not depend on each other and none of the primitives benefit from using all cores for itself. However, this is not exposed in the user-facing Accelerate language.

[71] The ADBench repository also includes files for $N = 2.5 \cdot 10^6$, but the framework does not support this size yet.

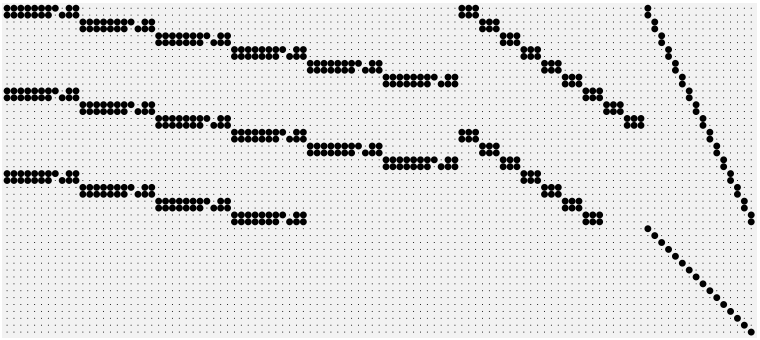


Figure 4: The (potentially) nonzero entries in a Jacobian for BA where $N = 6$, $M = 9$, $P = 16$. Note the horizontal division into the camera parameters, the points and the weights, and the vertical division into the reprojection errors and the weight errors. The holes in the "$\frac{\partial \text{reproj\_error}}{\partial \text{camera}}$" blocks are there because the elements of $\boldsymbol{x}_0$ only contribute to one of the two outputs of reprojerr(). This figure is inspired by a figure in [51].

$$
\begin{aligned}
(N, M, P) \in \{&(49, 7776, 31843), && (21, 11315, 36455), && (161, 48126, 182072),\\
&(372, 47423, 204472), && (257, 65132, 225911), && (539, 65220, 277273),\\
&(93, 61203, 287451), && (88, 64298, 383937), && (810, 88814, 393775),\\
&(1197, 126327, 563734), && (1723, 156502, 678718), && (253, 163691, 899155),\\
&(245, 198739, 1091386), && (356, 226730, 1255268), && (1102, 780462, 4052340),\\
&(1544, 942409, 4750193), && (1778, 993923, 5001946), && (1936, 649673, 5213733),\\
&(4585, 1324582, 9125125), && (13682, 4456117, 2987644)\}
\end{aligned}
$$

Figure 5: Size parameters for the 20 BA input files in the ADBench suite.

For BA, there are 20 input files with increasing sizes, with parameters shown in Fig. 5.

For both GMM and BA, for each input file, we run our implementation 10 times, or until the total execution time has reached 10 seconds, whichever occurs sooner. This strategy is also taken by a number of the existing implementations in the ADBench suite. If this results in a total runtime for a single input file that is longer than 900 seconds, our program is considered to time out on that input file. This time-out limit of 900 seconds is global, and also applies to the benchmark implementations in other languages.

In practice, the described testing strategy results in some variance for some of the test cases; however, this is not too much to be able to use the results. To check this, we ran all tested benchmark implementations twice on all input files. For each implementation (including the non-Accelerate ones) and input file, if the implementation did not time out when run on that input file, we collected the ratio between the second gradient computation time and the first gradient computation time. Figure 6 shows a histogram of these time ratios. All measurements differ by at most a factor 1.73; 98% of the measurements differ by less than a factor of 1.2. As we will see in Section 5.3, a variance of 1.2 is insignificant relative to the difference in performance between the various implementations of the benchmark tasks.

Programs. In addition to our own implementations of GMM and BA, we also ran a number of the existing implementations of those tasks in the ADBench suite. The selection was based purely on which ones were easy to get running on our testing machine. The implementations in the ADBench suite that did not work for us despite an indication to the contrary in the documentation, are DiffSharp (.NET) and Julia's Zygote library. All others are included in our benchmark.

We build on the ADBench framework as of September 24th, 2020 (commit `e30b6c70`), and the other implementations in the benchmark are as included in the repository at that point in time.

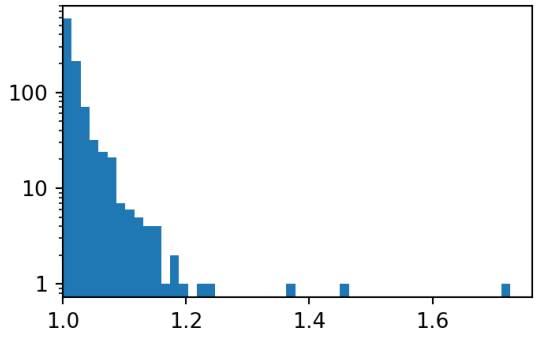


Figure 6: Histogram of the ratio of the gradient computation time between two runs of the full benchmark suite for all tested implementations. The histogram counts $\max\{r, \frac{1}{r}\}$ for all ratios r to uniformise the direction of bias.

5.2.1 Experimental setup

The specifications of the machine on which the performance tests were run, are as follows:

- OS: Ubuntu 20.04.1 LTS (Linux 5.4.0)
- Processor: AMD Ryzen Threadripper 2950X at 3.5GHz (16 physical cores, 32 threads)
- Memory: 64GiB (with 64GiB swap)
- Versions: GHC 8.8.4; code based on Accelerate 1.3.0.0 with Stackage LTS 16.17
- Compilation: `ghc -O2 -threaded -rtsopts`; runtime flags: `+RTS -M64G -qg`

Remarks:

- The `-M64G` runtime flag limits the total Haskell heap size (which includes any Accelerate array memory allocations) to 64GiB, which is the total amount of RAM on the test system. Due to GC effects and system tasks taking a small fraction of memory, we cannot limit memory usage to exactly the available RAM; therefore, we limit memory to approximately the available RAM. We note that using swap memory is severely detrimental to performance, which we thus want to avoid.
- The `-qg` runtime flag forces the Haskell GC to run single-threadedly. We measured the difference on all test cases between the standard multi-threaded setup and the single-threaded GC with `-qg`, and determined that with `-qg`, gradient computation time rose by more than 10% on only four test cases, whereas it fell by more than 10% on 41 test cases (16% of the total of 250 test cases between GMM and BA). Overall, gradient computation time generally decreased somewhat.

5.3 Results

The methodology in Section 5.2 was applied to compare a number of implementations of the GMM and BA tasks. As mentioned previously, the GMM implementations were run on the input files for $N \in \{10^3, 10^4\}$ and the BA implementations were run on all 20 input files.

In Appendix D we present three graphs corresponding to these three sets of input files (two for GMM and one for BA). Our implementations are plotted with a dashed line in order for them to stand out more; the dashed line style does not have any other meaning.

Both axes of the graph are scaled logarithmically. While this means that different time complexity translates to merely a different slope of the line, it allows packing the full, diverse set of

implementations and input sizes in one graph. In order to be able to judge time complexity more effectively, the plots include faint gray lines with slope 1, vertically separated by a factor $\sqrt[4]{10}$. A plot line that is parallel to these slope-1 lines indicates linear time complexity on the tested domain; faster or slower ascent indicates, respectively, super- or sub-linear time complexity on the tested domain.

Each implementation has a line, and each test case has a point on that line. The horizontal axis is the number of independent variables in the input: this is the number of variables that the gradient is taken with respect to. This measure is one way, albeit for some implementations an imperfect way, to measure the difficulty of the input. The vertical axis is the number of seconds taken to compute the gradient (or Jacobian, in the case of BA), measured as explained in Section 5.2. The time taken is measured as wall-clock time, not CPU time. Recall that a time limit of 900 seconds is imposed on the total execution time of an implementation, and that additionally an implementation can run out of the available RAM on the system; truncation of the graph of an implementation means that one of these things happened.

5.3.1 Four versions

Shown in the graphs in Appendix D, there are four Accelerate-based versions of our implementation for GMM and two for BA. Each of the four versions shares the same Accelerate code; the difference lies in the compilation and execution settings.

Firstly, the orange graphs use the full 32-wide parallelism of the machine—the default configuration of Accelerate. The blue graphs instead use only one CPU thread for the computation. This version distinction is present both for GMM and for BA.

Secondly, the cross-marked graphs faithfully use the reverse AD algorithm as described in Sections 3 and 4, while the pentagon-marked graphs (marked "recompute" in the legend) apply a modification to the algorithm that works out in practice to be an optimisation for GMM. As we already mentioned in passing earlier in this thesis, the algorithm as described tries to avoid as much recomputation as possible by storing all intermediate values in the primal pass. The memory usage (and memory traffic) caused by this choice can get out of hand in practice: in particular, storing the temporaries tuple from expression AD (Section 3.4.1) requires a large amount of memory to save a bit of computation that is usually very cheap. To prevent this effect, for the implementation versions marked "recompute" we change the primal builder for `Map` in Table 1:

```
Alet an_res = Map P(fun) (Avar am)
in Alet an = Map λfst (Avar an_res)
in Alet an_tmp = Map λsnd (Avar an_res)
in _
```

to the following:

```
Alet an = Map λfst (Map P(fun) (Avar am))
in Alet an_tmp = Map λsnd (Map P(fun) (Avar am))
in _
```

and we perform the analogous change to the primal builders of `Generate` and `ZipWith` in Table 1.

The reason why this change is equivalent to recomputing the primal expression temporaries in the dual pass is the array fusion optimisation performed by the Accelerate compiler. Assume for this discussion that the `Map` in question is used only once in the original program to be differentiated. If that is true, an and an`tmp` are used exactly once in the gradient program, both before and after the builder change described here. The difference imparted by the change is the following. Before (in the non-"recompute" versions), the value an`res` is shared and thus will

not be fused into its usage points.[72] After (with "recompute"), the two `Map` $\mathcal{P}$(`fun`) (`Avar` am) programs are both used only once[73], meaning that they can be fused into the `Map`s that define an and an`tmp` and, subsequently, into their respective usage points. The usage point of an`tmp` is the `ZipWith` running $\mathcal{D}$(`fun`) in the dual pass (as can be seen in Table 2); therefore, fusing an`tmp` (*including* the computation of $\mathcal{P}$(`fun`)) into its usage point means recomputing the primal expression temporaries in the dual pass.

The "recompute" change has no effect on the BA implementation, because BA does not use array AD but rather only expression AD; therefore, for BA, we show only two implementation versions: multi-threaded and single-threaded.

5.3.2 Conclusions

Having described what the graphs in Appendix D show, we can now start interpreting those results. We first discuss overall results, and afterwards we discuss the specific differences between the four versions of our implementation.

Time complexity. Most plot lines in all three graphs tend towards being parallel to the slope-1 background lines, and some already do so for small input sizes. These implementations have, at least as the input size gets larger, approximately linear time complexity in the size of their input. Since the objective function is itself linear in both GMM and BA, this means that those implementations have the correct time complexity for reverse AD: one of the prime characteristics of reverse AD is that the gradient function may only add a constant factor to the running time as compared to the original function.

There are two kinds of (partial) exceptions to this rule: plot lines that appear to indicate super-linear behaviour, and analogously sub-linear behaviour.

Super-linear behaviour we find only in the Julia and C++ Finite implementations for GMM; all other plot lines (eventually) become approximately linear. The exception for Julia and C++ Finite is to be expected: these two programs do not implement reverse AD, but instead compute the required gradient using finite differencing (see e.g. Section 4.3). Finite differencing adds a factor linear in the size of the input to the running time of the program, bringing the linear objective function to quadratic complexity. Notable is that this (approximate) quadratic behaviour does not occur for BA; this is because in the BA task, the input size merely adds extra parallelism to the task without increasing the number of independent variables in one particular gradient computation (which is always 15 or 1 depending on the Jacobian block; see Section 5.1).

Partial sub-linear behaviour we find for implementations that make better use of the parallelism offered by the machine as the input size gets larger. We first focus on GMM, and afterwards we look at the effects on the BA task.

Increasing parallelism allows a program to increase the amount of computation performed in a single unit of time, which makes it possible to deal with larger input sizes without linearly increasing execution time. The resulting sub-linear behaviour we see most strongly with PyTorch, TorchScript, TensorFlow Eager and multi-threaded Accelerate on the GMM task. Clearly, most of the the tested input sizes do not exhaust the capability of these implementations to parallelise their computations; in particular, even if PyTorch and TorchScript are relatively slow for the smallest input sizes, they are easily the fastest implementations at the largest input sizes. The

[72] This is a design decision in the current array fusion algorithm in Accelerate. Should the Accelerate compiler apply more program intelligence here in the future, the necessity of the described "recompute" change may well disappear.

[73] Note that the Accelerate compiler currently does not perform common-subexpression elimination (CSE); for user programs, CSE is done by GHC before the Accelerate compiler starts processing the program.

sub-linear behaviour can be seen to a lesser extent in TensorFlow Graph, which starts out faster than PyTorch on the GMM task but ends up being somewhat slower on the largest cases.

The multi-threaded versions of our Accelerate implementation also show suboptimal performance on smaller cases, but significantly more respectable timings for the larger GMM cases. For the largest tested GMM cases, we approach a factor $\sqrt{10} \approx 3$ above TensorFlow Graph and PyTorch. However, we do use much more resources: our plot lines truncate early on both GMM graphs due to exhaustion of the available RAM. Furthermore, Accelerate always use all available threads on the machine, while PyTorch and TensorFlow are much more conservative and appear to use only what they need.

For BA, we see linear behaviour in all implementations except our Accelerate programs, which—for the multi-threaded version as well as, to a lesser extent, the single-threaded version—show sub-linear behaviour again. In general, it is clear that Accelerate has a higher start-up cost than many other implementations, but in return can sometimes make better use of the available parallelism for larger inputs.

Absolute performance. Immediately evident upon perusing the graphs is the suboptimal performance of Autograd on both tasks, and of TensorFlow Graph and PyTorch on BA. This is consistent with the results reported by the authors of the ADBench suite.[74]

The manual C++ implementations (which do not use AD, but result from manual differentiation of the functions involved) have the most consistent good performance, but due to their poor parallelisation on GMM, they are overtaken by PyTorch on that task for larger inputs. The finite-differencing-based implementations (Julia and C++ Finite) have low overhead on small cases, but become very slow on larger cases, as expected.

On GMM, Accelerate is generally slower than the good implementations, but manages to close most of the gap for inputs where it can use the parallelism of the machine. For BA, the parallelism is easier to exploit, and Accelerate quickly becomes the fastest implementation in our benchmark, although it suffers heavily from excessive RAM usage leading to an early cutoff of the plot line.

Our implementation. On GMM, the normal and the "recompute" versions have generally similar performance profiles, with the version that recomputes expression temporaries having better absolute performance across the input domain. Recomputing expression temporaries additionally helps reduce the required RAM for Accelerate: this is to be expected, since less data is being stored between the primal and dual phases. Besides being somewhat more efficient overall, the "recompute" versions therefore also continue to succeed for larger input sizes.

From comparison of the multi-threaded and single-threaded versions on GMM, we see that using multiple threads has significant overhead on smaller cases, due to which the single-threaded version is significantly more performant there. While the multi-threaded version has an advantage on large input sizes in our benchmark, we hypothesise that on a machine with less parallelism than our relatively high-end benchmarking machine, this advantage will be significantly less.

The results on BA are much cleaner than on GMM, with multi-threaded Accelerate being faster than the single-threaded version on all inputs, and increasing that advantage as more data-parallelism can be exploited for larger cases.

[74] See https://adbenchwebviewer.azurewebsites.net/Plots/GetStatic?dir=2020-11-15_15-17-39_4e856a4c467522673c27320d38a7cba19170231c/. Their graphs additionally include Julia Zygote and DiffSharp in the gap between the faster implementations and the slowest ones.

5.3.3 Discussion

Firstly, a limitation of our benchmark is that some of the tested implementations, such as TensorFlow and PyTorch, may perform even better when run on massively parallel hardware like a GPU. For the reasons mentioned earlier, we have not tested GPU performance for any of the implementations for this thesis report.

Running a program transformed using our reverse AD implementation using Accelerate on only one thread, even if the machine has more available compute parallelism, has an advantage on smaller input sizes if the problem is complicated enough to not be *embarrassingly parallel*. This is probably because the Accelerate runtime scheduler, in addition to having to start the compute threads in the first place, has a certain amount of overhead that a small workload cannot overcome. However, even for the case of GMM, where the output program after our reverse AD transformation and after array fusion still contains 282 primitives,[75] Accelerate manages to use hardware parallelism to speed up execution significantly.

Recomputing expression temporaries saves work and memory on GMM and is an overall effective optimisation. Notable is that these savings already occur on multicore CPU, where memory access is relatively cheap compared to a GPU; we hypothesise that when running on a GPU platform, the savings here will be even larger. Indeed, especially for GPU targets where some additional parallel computation is generally cheap, we believe that the checkpointing optimisation for an AD transformation (see Section 2.1.4) may prove to be especially effective, both to reduce maximum memory usage and to reduce memory traffic.

With the limited benchmarking that we have done, we cannot draw unconditional conclusions about the performance of our reverse AD transformation. However, even so, we believe to have shown that our algorithm is capable of producing reasonably performant code when paired with the existing performance work that has gone into the Accelerate compiler. To be able to compete with the best-in-class, improvements are still necessary; an important, but not the only, point of improvement is the memory usage of the output programs of the AD transformation.

[75] Not counting delayed (non-instantiated) `generate` expressions. This number 282 is significantly higher than for typical Accelerate programs due to the blow-up resulting from our reverse AD transformation, which also inhibits some array fusion that may otherwise have been possible in the original program (see also Section 6.1). The number 282 may decrease in the future, even without modifications to the reverse AD implementation itself, by addition of new program optimisations in the Accelerate compiler and improvements to the array fusion engine.

6 Conclusions

In the introduction to this thesis, we explained how Bayesian inference algorithms and machine learning algorithms need a high-performance parallel processing framework in order to easily express algorithms in an efficient way. We also argued how functional parallel array programming languages answer this need, assuming that they provide reverse-mode automatic differentiation functionality. Few such languages provide this functionality at the moment, and to our knowledge, there is as of yet no mature and high-performance implementation of reverse-mode AD for a functional parallel array language (Section 6.2).

One such programming language is Accelerate [8], and in this thesis project we present an implementation of reverse-mode AD for a significant subset of the Accelerate language. We do this by first defining a source-code transformation for reverse AD on an Accelerate-like language (Section 3), which covers most of the features present in the full Accelerate language with the exception of loops and some of the most involved array primitives (Section 3.2.1). Furthermore, we present an implementation of this algorithm in the Accelerate compiler, which uses a type-indexed AST in order to statically prevent type-safety issues in the output of the source-code transformation (Section 4). Benchmarks show that the produced output programs perform reasonably well when compared with existing implementations for the same benchmark tasks in other languages (Section 5).

In the introduction, we asked some questions to guide our research project, to which we can now provide some partial answers.

1. What additional language primitives, on top of those in the current Accelerate language, are needed to express the gradient computation code for an Accelerate program?

   Our reverse AD algorithm does not yet handle the full Accelerate language, so we cannot give a full answer to this question. However, for the subset of the language that we do support, the derivative can be expressed within (the entirety of) Accelerate: the extra components we need in the output language that our input language does not contain, are left and right scans (for the derivative of `Foldl`), and `Permute` (for the derivative of `Backpermute` and array indexing).

   In order to support loops in the input language, we expect to need an extension to the Accelerate language to be able to incrementally fill an array with primal values. For the yet unsupported array primitives like `Permute`, the existing Accelerate language may well be sufficient, but this is still unclear.

2. How does one perform non-local full-program transformations (in particular, reverse-mode AD) on a strongly-typed AST?

   Because of the non-local nature of a reverse AD transformation that is organised like ours, type-correctness of such a transformation is non-trivial to prove, especially in the Haskell type system. Our implementation shows one method for circumventing this problem: by using runtime type checking, we defer to runtime certain components of the type-correctness proof that are hard to give statically. In this way, we complete the proof dynamically for each input program that the compiler receives, crashing in case the proof should fail at runtime. While suboptimal from a theoretical standpoint, runtime type checking does allow for a pragmatic implementation of the algorithm without compromising the type-safety of the differentiated Accelerate program.

3. What optimisations, both generic compiler optimisations and improvements to the AD algorithm, are necessary for competitive performance on top of a naive implementation of AD?

While we showed that one improvement to our AD algorithm has a positive effect on our benchmarks, namely recomputing expression temporaries in the dual pass (Section 5), we have unfortunately been unable to research this topic more thoroughly. Nevertheless, thanks to the optimisations already implemented in the current Accelerate compiler and the LLVM backend that it uses for scalar-level computations, our implementation of reverse AD can already produce reasonably performant code. This is true in particular when one considers the difference in the amount of research invested in our algorithm versus established AD implementations like PyTorch and TensorFlow.

In the main research question formulated in the introduction to this thesis, we asked how a reverse-mode AD system for Accelerate using source-code transformation can provide performance competitive with existing fast AD frameworks. While the method we describe in this thesis cannot yet compete with the state-of-the-art in the field of AD, we believe that our pragmatic approach to program differentiation has promise for producing an algorithm that can challenge the existing options.

In the sections below, we will discuss some general remarks about our algorithm, as well as various potential points of improvement and further research. Afterwards, we will briefly compare our work with existing algorithms and implementations of reverse AD, some of which have already come up in the discussion of background in Section 2.

6.1 Discussion & Future work

There are some general remarks that we still wish to make regarding our reverse AD algorithm and its implementation. Many of these remarks hint at new questions and unknowns; most research projects produce more questions than they give answers, and this thesis work is no exception.

Second-order representation. In the design of our reverse AD algorithm, we are helped significantly by the second-order nature of the language we work on (Idealised Accelerate, modelling Accelerate). As argued in Section 2.3.2, a second-order language is easier to compile to GPU code than a more generic, higher-order language; however, the two-part structure of our AD algorithm (split in the array and expression halves) is also due to this second-order nature. Our array AD algorithm is essentially (and was constructed as) a generalised version of the algorithm for expressions, even if our exposition was the other way round.

Functional array languages need not necessarily be second-order for our algorithm to apply, however. Indeed, when compiling to GPU code, one has to represent the program in something resembling our second-order structure at some point in the pipeline, due to the GPU programming model. If a language with a different user-facing structure at some point compiles down to an internal second-order language, and that internal language is still purely functional, our algorithm may be adaptable to run at that point in the compilation pipeline.

However, the two-way split of our algorithm also raises questions: can elements of our algorithm generalise to languages too expressive to easily compile down to a second-order functional array language? Examples of such languages might be ones that support exotic control flow or low-level operations that are hard to express in the rather simple (programming-language-theoretically), pure functional language that this thesis works on.

Fanout. A general principle in reverse AD is that fanout in the computation graph of the input function translates to addition in the dual. Fanout in Idealised Accelerate occurs in two places: sharing introduced with let-bindings (both on the array and the expression level), and parallel usage of values in a parallel array primitive.

These two types of fanout we handle in different ways in our algorithm. Let-induced sharing we detect and reify on the fly using the contribution map passed through the algorithm. Parallel usage of values in an array primitive we handle differently based on whether the parallel usage is inherent in the array primitive, or introduced using array indexing in a user-specified expression. For example, the `Replicate` primitive dualises to a construction that sums the adjoint contributions arising from sharing using a `Sum` primitive, after shuffling those together in the innermost dimension. For `Backpermute` as well as sharing introduced by array indexing, the `(+)` combination function in the generated `Permute` primitive adds adjoints and dualises the fanout there.

Being able to concretely pinpoint where sharing occurs seems important in designing an AD algorithm. Another approach to making sharing explicit is requiring the use of a duplication combinator in a combinator-based program representation (e.g. [17]). This explicitness is not always attained or required: for example, in [58], mutation (addition to accumulator cells) is used to resolve sharing at runtime. Can we still do reverse AD with as codomain a purely functional language if we cannot statically foresee all possible sharing in a program, for instance in a language with control flow too dynamic to fully analyse statically?[76] The absence of mutation in a program makes analysing the program (and its generating algorithm) for correctness easier, and often facilitates more invasive compiler optimisations. However, we do not know whether, for languages where such a transformation is significantly more difficult than one that introduces mutation, it is worth the additional effort to try.

Inefficient addition of array adjoints. In our array AD algorithm, our method for adding adjoint contributions involves generating an array of the required size using `Generate`, where each cell is individually computed to be the sum of only those contributions that are actually in-bounds for the various arrays that we receive. While it may sometimes be possible to guarantee that the adjoint contribution arrays are full-size, allowing us to elide the conditionals in the expressions, this will not always be the case. Since conditional execution is often suboptimal on very wide vector architectures like modern GPUs, it may be worth trying to reduce the number of times the addition of adjoint contributions really needs conditionals.

It is possible to change our algorithm to not generate empty arrays for intermediate adjoints in untaken conditional array branches, but instead generate arrays containing zeros; it is also possible to change the dual of `ZipWith` to ensure that the adjoint contributions it produces are always of the correct, full size. With both of these changes, the complicated addition of contributions using conditionals should not be necessary anymore. However, it is unclear whether both of these changes are worth the accompanying additional computation cost elsewhere, in particular generating full-size arrays of zeros for array conditionals. This may also depend on whether the resulting unnecessary arrays can be inspected and possibly elided by an optimising compiler; the Accelerate compiler cannot currently "see through" a conditional very well, and thus would not be able to perform such an optimisation.

Optimisation before AD. Suppose the user wishes to differentiate an Accelerate program that contains the following piece of code:

```
1 map (\x -> 2 * x) (map (\x -> x + 3) a)
```

With our current implementation, the array AD algorithm sees two `Map` primitives and differentiates them separately, generating a number of `Map` and `ZipWith` primitives and introducing sharing between these values along the way. If array fusion would have run *before* our AD algorithm, the two maps would have been combined, and the output code from AD would have been significantly simpler.

[76] Do we even want such languages?

In this case, the unnecessary complexity that we produce can still be optimised away using array fusion, but there may be more complicated cases where redundancy or verbosity in the original program is mangled too much by the AD pass for the optimiser to be able to understand afterwards. It could be very promising to perform some preliminary optimisations on the AST *before* running the AD algorithm, while still running the full suite of optimisations after AD. This may (or may not) increase compile times, but may also lead to better code being generated.

Reducing sharing. As already mentioned in the previous point and in Section 5.3.3, our array AD algorithm increases the program size significantly, in addition to introducing additional sharing that was not present in the original program. Empirically, this prevents array fusion from being able to do much on the output from AD, at least as compared to hand-written Accelerate programs.

One way to improve this situation might be to significantly improve the intelligence of array fusion, so that it is able to change the program more drastically than it currently does. On the other hand, it may also be worthwhile to avoid generating so much sharing in the first place. Some sharing is certainly needed—the core of reverse AD is to remember computed values from the primal pass in the dual pass, and for this reason those values are by necessity used at least twice. However, checkpointing (Section 2.1.4) can reduce the amount of values retained from the primal pass, and furthermore it may well be possible to optimise the generated code in Tables 1 and 2 to help reduce the magnitude of the problem.

Reducing sharing and the accompanying retention of data in memory may also help reducing the excessive use of memory that we observed about our implementation in Section 5.

Additional optimisations. The code produced by our AD transformation contains patterns not usually found in hand-written programs, and these patterns may benefit from tailored, additional optimisation passes in the compiler. In particular, our policy of storing all primal values for the dual pass sometimes results in redundant stores. Simple unused array stores are easily removed using dead-code elimination, but on the expression level, removing redundant primal stores would require detecting that a particular element of the tuple values in an array is never actually used, and subsequently applying a type-changing transformation on the program that removes that tuple element. For expression code, we also regularly store the same value multiple times (since multiple expression nodes, like `Evar` nodes, may refer to the same value). Removing that redundancy could be done in a similar way with a program-wide optimisation.

The way we avoided the problem of redundant stores in our experiments (Section 5) was by recomputing all expression temporaries; this is unlikely to be optimal in all cases, especially if we start supporting loops in expression code.

An alternative, less drastic way to reduce the problem would be to carefully trim the to-store sets in our algorithm. However, we predict that this method will not be able to find all opportunities for store elimination, due to its local-only nature. Literature approaches the problem using a to-be-recorded analysis [23], which attempts to estimate the set of really required stores using a data-flow analysis on the original source program. We hypothesise that with a sufficiently intelligent dead-store analysis as described above, such a data-flow analysis may not be required, and we get a nice compiler optimisation as a side-benefit.

Special cases. The Accelerate language already contains some specialisation for special cases, for example by providing `Replicate` while the language already has `Backpermute` and `Generate` (see also Section 2.3.1). However, for reverse AD, many more special cases exist that do not currently have a specialised primitive but would nevertheless benefit from a special-cased derivative implementation.

A prominent example of this effect is for `Backpermute` operations, or other primitives containing array indexing, where the formula determining the index into the indexed array is (easily) invertible. For example, consider the reversal of a single-dimensional array, or taking the transpose of a square two-dimensional array. These operations can be easily implemented using either `Backpermute` or `Generate` (in the latter case using array indexing); the algorithm in Section 3 implements the derivative of both constructions using a `Permute`. However, the derivatives of both array reverse and array transpose happen to be the operations themselves, which do not need a relatively costly `Permute` but can instead simply be a `Backpermute`: not only does a `Permute` have a less-efficient parallel implementation, it also admits less opportunities for array fusion than a `Backpermute`.

Benchmarks. Less related to the algorithm and more to our methodology, the benchmarking we did to evaluate the performance of our implementation in Section 5 is very limited. Indeed, we evaluated only one program using array-level reverse AD. To better understand how our algorithm performs in different settings than the ones we tested, more extensive benchmarking is needed. Furthermore, benchmarking on a GPU is missing from our experiments.

6.2 Related work

Some of the work we mention here has already been discussed in Section 2; in those cases, we refer back to the relevant earlier subsection. This section of related work is placed at the end of the thesis report in order to be able to compare with the method we presented here.

- Multiple potential methods exist for performing reverse AD on a higher-order functional language. Pearlmutter and Siskind [42] describe an algorithm for doing reverse AD directly on an untyped lambda calculus (already discussed in Section 2.2.1), while Elliott [18] and Vákár [55] instead convert the program to a combinator-based language (Section 2.2.3), where sharing is made explicit using duplication combinators, before performing reverse AD there.

  While all these methods are very general, they are in a sense too general for our work on Accelerate: while an Accelerate program could theoretically be converted into the respective input languages for any of the above approaches, the output from reverse AD would then need to be converted back from a higher-order language to the second-order Accelerate language. Doing this conversion in general is relatively hard, and may in some cases not even be possible to do in a performant way. However, properties of the mentioned reverse AD algorithms (especially that in [55]) may imply that for a suitably restricted input language, the output can be converted to a second-order language more easily than it would be in general; we did not pursue this avenue of research.

- In contrast to the methods mentioned in the previous point, the technique described by Shaikhha et al. in [49] (discussed in Section 2.2.4) is defined on a functional array language and would therefore be much more easily applicable to Accelerate. However, due to their choice of doing a normal forward AD transformation first and subsequently optimising the program using intelligent compiler optimisations, it is very much unclear for what languages the result has the correct time complexity to be called reverse AD. While we have presented no more proof that our method has the correct complexity, we do believe that a hypothetical proof for our method can use much more local reasoning than a similar proof for the method in [49] would.

- In [58], Wang et al. describe an algorithm for reverse AD on a functional language augmented with mutable state, and they use a continuation-passing style (CPS) transform to eliminate the control operators they introduce. While attractive in its relative simplicity, this method is hard to apply to Accelerate for the same reasons as the higher-order methods above are: even for a purely functional input program, the algorithm (including the CPS transform)

produces an output program that uses mutation, and mutation is very hard to simulate in Accelerate. Traditional store-passing, as also described in [58, §3.6], uses an immutable map to keep track of state, but Accelerate does not support such a data structure in a reasonable way.

- There is some ongoing work to implement a reverse AD transformation for purely functional array languages outside of the present project; we know of relevant work on the Futhark language (yet unpublished) and the Dex language [32] (unpublished, but a partial implementation is available[77]). At the time of writing, neither of these projects uses a fully type-indexed AST like Accelerate does.
- Finally, there is a large body of research, and a large number of implementations, focusing on reverse AD for imperative languages. For these, we refer to more generic surveys of AD, for example [5, 35].

[77] https://github.com/google-research/dex-lang

Bibliography

[1] 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.

[2] Robert Atkey, Sam Lindley, and Jeremy Yallop. Unembedding domain-specific languages. In Stephanie Weirich, editor, *Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell, Haskell 2009, Edinburgh, Scotland, UK, 3 September 2009*, pages 37–48. ACM, 2009.

[3] Lennart Augustsson and Kent Petersson. Silly type families. Available at https://web.cecs.pdx.edu/~sheard/papers/silly.pdf, 09 1994.

[4] Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, James Bergstra, Ian Goodfellow, Arnaud Bergeron, Nicolas Bouchard, David Warde-Farley, and Yoshua Bengio. Theano: new features and speed improvements. In *NIPS 2012 Workshop: Deep Learning and Unsupervised Feature Learning*, 2012.

[5] Atilim Gunes 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.

[6] Nick Benton, Chung-Kil Hur, Andrew J Kennedy, and Conor McBride. Strongly typed term representations in Coq. *Journal of automated reasoning*, 49(2):141–159, 2012.

[7] Bob Carpenter, Matthew D. Hoffman, Marcus Brubaker, Daniel D. Lee, Peter Li, and Michael Betancourt. The Stan Math library: Reverse-mode automatic differentiation in C++. *CoRR*, abs/1509.07164, 2015.

[8] 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. ACM, 2011.

[9] Siddhartha Chatterjee, Guy E. Blelloch, and Marco Zagha. Scan primitives for vector computers. In Joanne L. Martin, Daniel V. Pryor, and Gary Montry, editors, *Proceedings Supercomputing '90, New York, NY, USA, November 12-16, 1990*, pages 666–675. IEEE Computer Society, 1990.

[10] Curtis Chin Jen Sem. Formalized correctness proofs of automatic differentiation in Coq. *Master's Thesis, Utrecht University*, 09 2020. https://dspace.library.uu.nl/handle/1874/400790; Coq code: https://github.com/crtschin/thesis.

[11] Robert Clifton-Everest, Trevor L. McDonell, Manuel M. T. Chakravarty, and Gabriele Keller. Streaming irregular arrays. In Iavor S. Diatchki, editor, *Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell, Oxford, United Kingdom, September 7-8, 2017*, pages 174–185. ACM, 2017.

[12] Accelerate contributors. `Data.Array.Accelerate` (accelerate-1.3.0.0). https://hackage.haskell.org/package/accelerate-1.3.0.0/docs/Data-Array-Accelerate.html. Accessed: 2020-11-28.

[13] Olivier Danvy and Andrzej Filinski. Abstracting control. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming, LFP 1990, Nice, France, 27-29 June 1990*, pages 151–160. ACM, 1990.

[14] Olivier Danvy and Lasse R. Nielsen. Defunctionalization at work. In *Proceedings of the 3rd ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 162–174, 2001.

[15] Nicolaas G. de Bruijn. Lambda-calculus notation with nameless dummies: a tool for automatic formula manipulation with application to the Church–Rosser theorem. *Indag. Math*, 34(5):381–392, 1972.

[16] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In Eric A. Brewer and Peter Chen, editors, *6th Symposium on Operating System Design and Implementation (OSDI 2004), San Francisco, California, USA, December 6-8, 2004*, pages 137–150. USENIX Association, 2004.

[17] Conal Elliott. Compiling to categories. *Proc. ACM Program. Lang.*, 1(ICFP):27:1–27:27, 2017.

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

[19] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In Robert Cartwright, editor, *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation (PLDI), Albuquerque, New Mexico, USA, June 23-25, 1993*, pages 237–247. ACM, 1993.

[20] Seth Flaxman, Swapnil Mishra, Axel Gandy, H. Juliette T. Unwin, Thomas A. Mellan, Helen Coupland, Charles Whittaker, Harrison Zhu, Tresnia Berah, Jeffrey W. Eaton, Mélodie Monod, Azra C. Ghani, Christl A. Donnelly, Steven Riley, Michaela A. C. Vollmer, Neil M. Ferguson, Lucy C. Okell, Samir Bhatt, and Imperial College COVID-19 Response Team. Estimating the effects of non-pharmaceutical interventions on COVID-19 in Europe. *Nature*, 584(7820):257–261, 2020.

[21] Martijn Fleuren. Independently computed regions in a data parallel array language. *Master's Thesis, Utrecht University*, 01 2020. https://dspace.library.uu.nl/handle/1874/395368.

[22] Andreas Griewank, David W. Juedes, and Jean Utke. Algorithm 755: ADOL-C: A package for the automatic differentiation of algorithms written in C/C++. *ACM Trans. Math. Softw.*, 22(2):131–167, 1996.

[23] Laurent Hascoët, Uwe Naumann, and Valérie Pascual. "To be recorded" analysis in reverse-mode automatic differentiation. *Future Generation Computer Systems*, 21(8):1401 – 1417, 2005.

[24] 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.

[25] Troels Henriksen. *Design and Implementation of the Futhark Programming Language*. PhD thesis, University of Copenhagen, Universitetsparken 5, 2100 København, 11 2017.

[26] Matthew D. Hoffman and Andrew Gelman. The No-U-Turn sampler: adaptively setting path lengths in Hamiltonian Monte Carlo. *J. Mach. Learn. Res.*, 15(1):1593–1623, 2014.

[27] Robin J. Hogan. Fast reverse-mode automatic differentiation using expression templates in C++. *ACM Trans. Math. Softw.*, 40(4):26:1–26:16, 2014.

[28] Aaron W. Hsu. *A data parallel compiler hosted on the GPU*. PhD thesis, Indiana University, 11 2019.

[29] Michael Innes. Don't unroll adjoint: Differentiating SSA-form programs. *CoRR*, abs/1810.07951, 2018.

[30] Alp Kucukelbir, Dustin Tran, Rajesh Ranganath, Andrew Gelman, and David M. Blei. Automatic differentiation variational inference. *J. Mach. Learn. Res.*, 18:14:1–14:45, 2017.

[31] Dougal Maclaurin. *Modeling, Inference and Optimization with Composable Differentiable Procedures*. PhD thesis, Harvard University, 4 2016.

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

[33] Frederik M. Madsen, Robert Clifton-Everest, Manuel M. T. Chakravarty, and Gabriele Keller. Functional array streams. In Tiark Rompf and Geoffrey Mainland, editors, *Proceedings of the 4th ACM SIGPLAN Workshop on Functional High-Performance Computing, FHPC@ICFP 2015, Vancouver, BC, Canada, September 3, 2015*, pages 23–34. ACM, 2015.

[34] Oleksandr Manzyuk, Barak A. Pearlmutter, Alexey Andreyevich Radul, David R. Rush, and Jeffrey Mark Siskind. Perturbation confusion in forward automatic differentiation of higher-order functions. *J. Funct. Program.*, 29:e12, 2019.

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

[36] Trevor L. McDonell, Manuel M. T. Chakravarty, Vinod Grover, and Ryan R. Newton. Type-safe runtime code generation: Accelerate to LLVM. In Ben Lippmeier, editor, *Proceedings of the 8th ACM SIGPLAN Symposium on Haskell, Haskell 2015, Vancouver, BC, Canada, September 3-4, 2015*, pages 201–212. ACM, 2015.

[37] 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.

[38] Trevor L. McDonell, Joshua D. Meredith, and Gabriele Keller. Embedded pattern matching. To be published; personal communication, 06 2020.

[39] Uwe Naumann. Optimal Jacobian accumulation is NP-complete. *Math. Program.*, 112(2):427–441, 2008.

[40] Radford M. Neal. MCMC using Hamiltonian dynamics, 2012.

[41] 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*, 2017.

[42] 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.

[43] Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In Richard L. Wexelblat, editor, *Proceedings of the ACM SIGPLAN'88 Conference on Programming Language Design and Implementation (PLDI), Atlanta, Georgia, USA, June 22-24, 1988*, pages 199–208. ACM, 1988.

[44] Matthew Pickering, Gergő Érdi, Simon Peyton Jones, and Richard A. Eisenberg. Pattern synonyms. In Geoffrey Mainland, editor, *Proceedings of the 9th International Symposium on Haskell, Haskell 2016, Nara, Japan, September 22-23, 2016*, pages 80–91. ACM, 2016.

[45] John C. Reynolds. The discoveries of continuations. *LISP Symb. Comput.*, 6(3-4):233–248, 1993.

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

[47] Andreas Schäfer and Dietmar Fey. High performance stencil code algorithms for GPGPUs. In Mitsuhisa Sato, Satoshi Matsuoka, Peter M. A. Sloot, G. Dick van Albada, and Jack J. Dongarra, editors, *Proceedings of the International Conference on Computational Science, ICCS 2011, Nanyang Technological University, Singapore, 1-3 June, 2011*, volume 4 of *Procedia Computer Science*, pages 2027–2036. Elsevier, 2011.

[48] Shubhabrata Sengupta, Mark J. Harris, Yao Zhang, and John D. Owens. Scan primitives for GPU computing. In Mark Segal and Timo Aila, editors, *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware 2007, San Diego, California, USA, August 4-5, 2007*, pages 97–106. Eurographics Association, 2007.

[49] 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.

[50] 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.

[51] Filip Šrajer, Zuzana Kukelova, and Andrew W. Fitzgibbon. A benchmark of selected algorithmic differentiation tools on some problems in computer vision and machine learning. *Optimization Methods and Software*, 33(4-6):889–906, 2018.

[52] 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.

[53] Bo Joel Svensson and Josef Svenningsson. Defunctionalizing push arrays. In Jost Berthold, Mary Sheeran, and Ryan Newton, editors, *Proceedings of the 3rd ACM SIGPLAN workshop on Functional high-performance computing, FHPC@ICFP 2014, Gothenburg, Sweden, September 4, 2014*, pages 43–52. ACM, 2014.

[54] Wouter Swierstra. Heterogeneous binary random-access lists. *J. Funct. Program.*, 30:e10, 2020.

[55] Matthijs Vákár. Reverse AD at higher types: Pure, principled and denotationally correct. In *Proc. ESOP 2021*. Springer, 2021. To appear.

[56] Dimitrios Vytiniotis, Stephanie Weirich, and Simon Peyton Jones. Simple unification-based type inference for GADTs. In *International Conference on Functional Programming (ICFP'06)*. ACM SIGPLAN, 04 2006. 2016 ACM SIGPLAN Most Influential ICFP Paper Award.

[57] Philip Wadler. Theorems for free! In Joseph E. Stoy, editor, *Proceedings of the fourth international conference on Functional programming languages and computer architecture, FPCA 1989, London, UK, September 11-13, 1989*, pages 347–359. ACM, 1989.

[58] 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.

A Unit test programs

For the notation used in this (multi-page) table, see Section 4.3.1. (* means finite differencing yields inaccuracies due to discontinuity of (the derivative of) the test program.)

Name	Inputs	Program
acond_0	$\forall a^V$	`sum (acond (the (sum a) > 0) a (map (*2) a))`
acond_0a	$\forall a^V$	`sum (acond (the (sum a) > 0) (map (*2) a) a)`

acond_1	$\forall a^V$	`let b = map (*2) a`
		`  T2 a1 _ = acond (the (sum a) > 0) (T2 a b) (T2 b a)`
		`in sum (map (\x -> x * toFloating (indexHead (shape a1))) b)`

*acond_2a	$\forall a^V$	`let b = map (2*) a`
		`  y = acond (the (sum a) <= 2) (map (+1) b) (map (subtract 1) (map (2*) b))`
		`in sum (zipWith (*) b y)`

acond_2a_friendly	$\forall a^{V\|\sum v-2\|>0.1}$	`let b = map (2*) a`
		`  y = acond (the (sum a) <= 2) (map (+1) b) (map (subtract 1) (map (2*) b))`
		`in sum (zipWith (*) b y)`

*acond_2b	$\forall a^V$	`let b = map (2*) a`
		`  y = acond (the (sum a) <= 2) (map (+1) b) (map (subtract 1) (map (2*) b))`
		`in sum (zipWith (*) y b)`

acond_2b_friendly	$\forall a^{V\|\sum v-2\|>0.1}$	`let b = map (2*) a`
		`  y = acond (the (sum a) <= 2) (map (+1) b) (map (subtract 1) (map (2*) b))`
		`in sum (zipWith (*) y b)`

acond_3a	$\forall a^V$	`sum (acond (constant True) (generate (I1 1) (\_ -> 42)) a)`

acond_3b	$\forall a^V$	`let I1 n = shape a`
		`in acond (n >= 0)`
		`  (generate I0 (\_ -> 42))`
		`  (zipWith (+) (sum (map (*2) a)) (sum (map (+3) a)))`

map	$\forall a^V$	`sum (map (*3) a)`
zipWith	$\forall a^V$ b^V	`sum (zipWith (\x y -> sin x * cos (x + y)) a b)`
fold_1	$\forall a^V$	`fold (*) 1 a`
fold_2	$\forall a^V$	`fold max 3 a`
*fold_3	$\forall a^V$	`fold (\x y -> let abs' v = cond (v < 0) (-v) v in abs' x + abs' y) 0 a`
fold_3_friendly	$\forall a^{V\|\text{allNonZero}(v)}$	`fold (\x y -> let abs' v = cond (v < 0) (-v) v in abs' x + abs' y) 0 a`
fold1_1	$\forall a^V$	`fold1 (*) a`
*fold1_2	$\forall a^V$	`fold1 max a`
fold1_2_friendly	$\forall a^{V\|\text{uniqueMax}(v)}$	`fold1 max a`

replicate_1	$\forall a^V$	`product . sum`
		`  . map (*2) . replicate (I2 (constant All) (5 :: Exp Int))`
		`  . map (\x -> x * x + x)`
		`  $ a`

replicate_2	$\forall a^V$	`fold1All (+)`
		`  . map (/20) . replicate (I2 (5 :: Exp Int) (constant All))`
		`  . map (\x -> x * x + x)`
		`  $ a`

replicate_3	$\forall a^V$	`fold1All (+) . map (*2)`
		`  . replicate (I4 (constant All) (5 :: Exp Int) (constant All) (3 :: Exp Int))`
		`  . map (\x -> x * x + x)`
		`  . replicate (I2 (constant All) (2 :: Exp Int))`
		`  $ a`

Name	Inputs	Program

replicate_4	$\forall \mathtt{a}^{V}$	`fold1All (+) . map (*2)`
		`  . replicate (I4 (5 :: Exp Int) (constant All) (3 :: Exp Int) (constant All))`
		`  . map (\x -> x * x + x)`
		`  . replicate (I2 (constant All) (2 :: Exp Int))`
		`  $ a`

slice_1	$\forall \mathtt{a}^{V}$	`let I1 n = shape a`
		`    m = n `div` 4 + 3`
		`    a1 = replicate (I3 m m (constant All)) a`
		`in sum (slice a1 (I3 (2 :: Exp Int) (1 :: Exp Int) (constant All)))`

slice_2	$\forall \mathtt{p1}^{[0..61]}$
	$\mathtt{p2}^{[0..61]}$
	$\mathtt{a}^{V}$
		`let I1 n = shape a`
		`    a1 = backpermute (I3 n n (n+1))`
		`                     (\(I3 i j k) -> I1 ((i + j + k) `mod` n))`
		`                     a`
		`in sum (slice a1 (I3 (constant p1 `mod` n) (constant p2 `mod` n) (constant All)))`

reshape_1	$\forall \mathtt{a}^{V\|_{\mathrm{even}(\#v)}}$	`let I1 n = shape a`
		`    b = reshape (I2 2 (n `div` 2)) a`
		`in product (sum b)`

reshape_2	$\forall \mathtt{a}^{V\|_{\mathrm{even}(\#v)}}$	`let I1 n = shape a`
		`    b = reshape (I2 (2 :: Exp Int) (n `div` 2)) a`
		`    c = reshape (I2 n 1) b`
		`in sum (product c)`

backpermute_1	$\forall \mathtt{m}^{[5..15]}$
	$\mathtt{a}^{V\|_{\#v>0}}$
		`let I1 n = shape a`
		`    b = backpermute (I2 (constant m) (2 * constant m))`
		`                    (\(I2 i j) -> I1 ((i + j) `mod` n))`
		`                    a`
		`in fold1All (+) b`

backpermute_2	$\forall \mathtt{m}^{[5..15]}$
	$\mathtt{a}^{V\|_{\#v>0}}$
		`let I1 n = shape a`
		`    b = backpermute (I2 (constant m) (2 * constant m))`
		`                    (\(I2 i j) -> I1 ((i + j) `mod` n))`
		`                    a`
		`    c = sum b`
		`    d = backpermute (I3 (2 :: Exp Int) (3 :: Exp Int) (constant m))`
		`                    (\(I3 _ i j) -> I1 (min (constant m - 1) (i * j)))`
		`                    c`
		`    e = map (/2) d`
		`    f = slice e (I3 (constant All) (constant All) (2 :: Exp Int))`
		`in fold1All (+) f`

aindex_generate_1	$\forall \mathtt{a}^{V}$	`let I1 n = shape a`
		`in sum (generate (I1 (n `div` 2))`
		`                    (\(I1 i) -> a ! I1 (2 * i)))`

aindex_generate_2	$\forall \mathtt{a}^{V}$	`let I1 n = shape a`
		`in sum (generate (I1 (2 * n))`
		`                    (\(I1 i) -> a ! I1 (i `div` 2) + a ! I1 (min (i `mod` 2) (n - 1))))`

aindex_generate_3	$\forall \mathtt{a}^{V}$	`let I1 n = shape a`
		`in sum (generate (I1 (2 * n))`
		`                    (\(I1 i) -> cond (i < 5) (a ! I1 (i `div` 2)) 42))`

aindex_generate_4	$\forall \mathtt{a}^{V}$	`let I1 n = shape a`
		`in sum (generate (I1 (2 * n))`
		`                    (\(I1 i) -> cond (i < n) (a ! I1 i) (a ! I1 (i - n))))`

aindex_map_1	$\forall \mathtt{a}^{V\|_{\#v\geq 3}}$	`sum (map (\x -> x + a ! I1 2) a)`

Name	Inputs	Program
*aindex_map_2	$\forall a^{V}$	`let I1 n = shape a` `in sum (map (\x -> x + a ! I1 (abs (round x) `mod` n)) a)`
aindex_map_2_friendly	$\forall a^{V\|_{\text{allNiceRound}(v)}}$	`let I1 n = shape a` `in sum (map (\x -> x + a ! I1 (abs (round x) `mod` n)) a)`

*aindex_map_3	$\forall a^{V}$	`let I1 n = shape a`
		`  b = fold max (-20)`
		`  (backpermute (I2 n n) (\(I2 i j) -> I1 (i .&. j)) a)`
		`in sum (map (\x -> x * b ! I1 (abs (round x) `mod` n)) a)`

*aindex_map_4	$\forall a^{V}$	`let I1 n = shape a`
		`  b = sum (backpermute (I2 n n)`
		`  (\(I2 i j) -> I1 (min (i `xor` j) (n-1)))`
		`  a)`
		`in sum (map (\x -> x * a ! I1 (abs (round x) `mod` n)) b)`

*aindex_acond	$\forall a^{V}$	`let I1 n = shape a`
		`  s = round (the (sum a))`
		`in sum (acond (0 <= s && s < n)`
		`  (generate (I1 1) (\(I1 _) -> a ! I1 s))`
		`  (generate (I1 1) (\(I1 _) -> 0)))`

aindex_only	$\forall a^{V}$	`let I1 n = shape a` `in sum (generate (I1 5) (\(I1 i) -> a ! I1 (i `mod` n)))`
a_ignore_argument	$\forall a^{V}$	`generate I0 (\_ -> 42.0)`
*nonfloat_lam	$\forall a^{V}$	`let b = map (\x -> T2 (round x :: Exp Int) (sin x)) a` `in sum (map (\(T2 i x) -> toFloating i * x) b)`
nonfloat_lam_friendly	$\forall a^{V\|_{\text{allNiceRound}(v)}}$	`let b = map (\x -> T2 (round x :: Exp Int) (sin x)) a` `in sum (map (\(T2 i x) -> toFloating i * x) b)`

logsumexp1	$\forall a^{V}$	`let I1 n = shape a`
		`  maxx = maximum a`
		`  shiftedx = zipWith (-) a (replicate (lift Any ::. n) maxx)`
		`in zipWith (+) (map log (sum (map exp shiftedx))) maxx`

*logsumexp2	$\forall a^{V}$	`let I1 n = shape a`
		`  maxx = maximum a`
		`  shiftedx = zipWith (-) a (replicate (lift Any ::. n) maxx)`
		`in map log (sum (map exp shiftedx))`

*logsumexp3	$\forall a^{V}$	`let maxx = maximum a` `in zipWith (+) maxx maxx`

tuple1	$\forall a^{V}$	`let T2 b _ = T2 (map (*2) a) (map (+3) a)`
		`  T3 a1 a2 a3 = T3 (zipWith (\x y -> sin x * y) b a) a b`
		`in map (+5) (sum (zipWith (\(T2 x y) z -> x + y * z) (zip a1 a2) a3))`

*tuple2	$\forall a^{V}$
	b^{V}
		`let T2 a1 a2 = T2 b (zipWith (*) a b)`
		`  T2 b1 b2 = T2 (zipWith (+) a2 b) a1`
		`  T2 c1 c2 = T2 b2 (zipWith (+) a1 b2)`
		`  T2 d1 d2 = T2 (sum c2) (sum b1)`
		`in zipWith (\x (T2 y z) -> log (y * z) + x) d1 (zip d2 (sum c1))`

Name	Inputs	Program

tuple3	$\forall a^{V\|\#v>0}$
	$b^{V\|\#v>0}$
		`let tupa@(T2 a1 a2) =`
		`  acond (let I1 n = shape a in n `mod` 3 == 0)`
		`  (T2 (replicate (I2 (constant All) (3 :: Exp Int)) b) a)`
		`  (T2 (generate (I2 1 2) (\(I2 i j) -> toFloating (i + j))) b)`
		`  T2 b1 b2 =`
		`  acond (let I2 n _ = shape a1 in cond (n > 0) (a1 ! I2 0 1) 42 > 0)`
		`  tupa`
		`  (T2 (backpermute (I2 4 4) (\(I2 i j) -> let I1 n = shape a2 in I1 (i * j `mod` n)) a2)`
		`  (slice a1 (I2 (constant All) (0 :: Exp Int))))`
		`in sum (zipWith (+) b2 (sum b1))`

tuple4	$\forall a^{V}$
	b^{V}
		`let T2 (T4 a1 s1 (T3 s2 a2 _) s3) a3 =`
		`  T2 (T4 b (sum a) (T3 (sum b) a (lift ())) (sum a)) b`
		`in zipWith (+) (zipWith (*) s1 (sum a3))`
		`  (zipWith (*) (zipWith (+) (zipWith (*) (sum a1) (product a2)) s2) s3)`

neural	$\forall len1^{[1..15]}$
	$len2^{[1..15]}$
	$input^{V\|\#v>0}$
	$weightdata^{V\|\#v\geq 3}$
		`let pickWeights :: (Shape sh, Elt a)`
		`  => Int -> Exp sh -> (Exp sh -> Exp DIM2) -> Acc (Vector a) -> Acc (Array sh a)`
		`  pickWeights seed size f arr =`
		`  let I1 n = shape arr`
		`  in backpermute size`
		`  (\idx -> let I2 i j = f idx`
		`  in I1 ((i + j + j * j * constant seed) ^ (1 + seed) `mod` n))`
		`  arr`
		`  mvmul mat v = let I2 m _ = shape mat`
		`  in sum (zipWith (*) mat (replicate (I2 m (constant All)) v))`
		`  dotp v1 v2 = sum (zipWith (*) v1 v2)`
		`  sigmoid v = map (\x -> 1 / (1 + exp (-x))) v`
		`  I1 inlen = shape input`
		`  w1 = pickWeights 1 (I2 (constant len1) inlen) id weightdata`
		`  w2 = pickWeights 2 (I2 (constant len2) (constant len1)) id weightdata`
		`  w3 = pickWeights 3 (I1 (constant len2)) (\(I1 i) -> I2 0 i) weightdata`
		`in sigmoid (dotp w3 (sigmoid (mvmul w2 (sigmoid (mvmul w1 input)))))`

*acond_cond1	$\forall a^{V\|\#v>0}$	`let b = map (*2) a`
		`  I1 n = shape a`
		`  sigmoid x = 1 / (1 + exp (-x))`
		`  -- idx is in [0, 2 * n)`
		`  idx = floor (sigmoid (the (sum a)) * (toFloating (2 * n) - 0.01)) :: Exp Int`
		`in -- This executes an out-of-bounds array index if one of the conditions is wrongly evaluated.`
		`  sum (acond (idx < n)`
		`  (generate (I1 (2 * n))`
		`  (\(I1 i) -> cond (i <= idx) (a ! I1 i) (b ! I1 ((i - idx) `div` 2))))`
		`  (generate (I1 (2 * n))`
		` (\(I1 i) -> a ! I1 (idx - n) + cond (i >= n) (b ! I1 (i - n)) (a ! I1 i))))

Name	Inputs	Program
*cond_0	$\forall a^V$	`sum (map (\x -> cond (x > 0) x (2 * x)) v)`
cond_0_friendly	$\forall a^{V\|_{\text{allNonZero}(v)}}$	`sum (map (\x -> cond (x > 0) x (2 * x)) v)`
cond_1	$\forall a^V$	`sum (map (\x -> 2 * cond (x > 0) (sin (x * x)) (x * x * exp x)) a)  -- derivative continuous at 0`

cond_2	$\forall a^V$	`let fn x =`
		`  let fmod u v = u - toFloating (floor (u / v) :: Exp Int) * v`
		`  y = cond (x `fmod` (2 * pi) < pi)  -- derivative continuous at all switching points`
		`  (sin x)`
		`  (pi / 12 * ((2 - 2 / pi * (x - pi / 2) `fmod` (2 * pi)) ^^ (6 :: Int) - 1))`
		`  in y * y`
		`in sum (map fn a)`

*cond_3a	$\forall a^V$	`sum (map (\x -> let u = 2 * x ; v = cond (x <= 2) (u + 1) (2 * u - 1) in u * v) a)`
*cond_3b	$\forall a^V$	`sum (map (\x -> let u = 2 * x ; v = cond (x <= 2) (u + 1) (2 * u - 1) in v * u) a)`

*cond_4	$\forall a^V$	`let fn x =`
		`  let u = cond (x >= 1) (log x + 1)  -- continuous`
		`  (cond (x - 2 < -3) (exp (x + 1) - 2) x)`
		`  y = 2 * exp u`
		`  in u * cond (u < (-1)) (u + y) y  -- not continuous`
		`in sum (map fn a)`

ignore_argument	$\forall a^V$	`sum (map (\_ -> 42.0) a)`

B ADBench task definitions

B.1 GMM: Gaussian Mixture Model fitting

Given:

- $N, D, K \in \mathbb{Z}_{>0}$
- $\boldsymbol{\alpha} = (\alpha_1, \ldots, \alpha_K) \in \mathbb{R}^K$
- $\boldsymbol{M} = (\boldsymbol{\mu}_1, \ldots, \boldsymbol{\mu}_K) \in \mathbb{R}^{K \times D}$
- $\boldsymbol{Q} = (\boldsymbol{q}_1, \ldots, \boldsymbol{q}_K) \in \mathbb{R}^{K \times D}$
- $\boldsymbol{L} = (\boldsymbol{\ell}_1, \ldots, \boldsymbol{\ell}_K) \in \mathbb{R}^{K \times \frac{D(D-1)}{2}}$
- $\boldsymbol{X} = (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N) \in \mathbb{R}^{N \times D}$
- $\gamma \in \mathbb{R}$, $m \in \mathbb{Z}$

Objective function: (highlighted are *scalar values* independent of varying parameters)

$$\begin{aligned} L(\boldsymbol{\alpha}, \boldsymbol{M}, \boldsymbol{Q}, \boldsymbol{L}) = & - \tfrac{1}{2} N D \log(2\pi) \\ & + \sum_{i=1}^{N} \operatorname{logsumexp}\left(\left[\alpha_k + \operatorname{sum}(\boldsymbol{q}_k) - \tfrac{1}{2} \left\| Q(\boldsymbol{q}_k, \boldsymbol{\ell}_k)(\boldsymbol{x}_i - \boldsymbol{\mu}_k) \right\|^2 \right]_{k=1}^{K} \right) \\ & - N \cdot \operatorname{logsumexp}(\boldsymbol{\alpha}) \\ & + \sum_{k=1}^{K} \left(\tfrac{1}{2}\gamma^2 \left(\|\exp(\boldsymbol{q}_k)\|^2 + \|\boldsymbol{l}_k\|^2 \right) - m \cdot \operatorname{sum}(\boldsymbol{q}_k) \right) \\ & - K \cdot \left(n' D \left(\log(\gamma) - \tfrac{1}{2}\log(2) \right) - \log \operatorname{MultiGamma}\left(\tfrac{1}{2} n', D \right) \right) \end{aligned}$$

where $n' = D + m + 1$, and $\operatorname{logsumexp}(\boldsymbol{x} : \mathbb{R}^n) = \log(\operatorname{sum}(\exp(\boldsymbol{x} - \max(\boldsymbol{x})))) + \max(\boldsymbol{x})$, and:

$$Q(\boldsymbol{q}, \boldsymbol{\ell}) = \begin{bmatrix} \exp(q_1) & 0 & \cdots & 0 \\ \ell_1 & \exp(q_2) & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ \ell_{D-1} & \ell_{D-1+D-2} & \cdots & \exp(q_D) \end{bmatrix}$$

and $\log \operatorname{MultiGamma}$ is SciPy's `scipy.special.multigammaln`[78], which can be defined by:

$$\begin{aligned} \log \operatorname{MultiGamma}(a, d) &= \log\left(\pi^{d(d-1)/4} \prod_{i=1}^{d} \Gamma(a - (i-1)/2) \right) \\ &= \tfrac{1}{4} d(d-1) \log(\pi) + \sum_{i=1}^{d} \log \Gamma(a - (i-1)/2) \end{aligned}$$

and $\log \Gamma$ is generally implemented in `lgamma` for real arguments.

The task is to compute the gradient of L with respect to its four arguments.

[78] https://github.com/scipy/scipy/blob/7a7843bc1b54968ee99796333743c73608c0638d/scipy/special/spfun_stats.py

B.2 BA: Bundle Adjustment

Input from file:

- $N, M, P \in \mathbb{Z}_{>0}$
- $onecam \in \mathbb{R}^{11}$; $onept \in \mathbb{R}^3$; $w \in \mathbb{R}$; $feat \in \mathbb{R}^2$

Precompute the following:

- $\boldsymbol{C} \in \mathbb{R}^{n \times 11}$; $\boldsymbol{C}[i][..] = onecam$
- $\boldsymbol{X} \in \mathbb{R}^{m \times 3}$; $\boldsymbol{X}[i][..] = onept$
- $\boldsymbol{w} \in \mathbb{R}^p$; $\boldsymbol{w}[i] = w$
- $\boldsymbol{F} \in \mathbb{R}^{p \times 2}$; $\boldsymbol{F}[i][..] = feat$
- $obs \in \mathbb{R}^{p \times 2}$; $obs[i] = [((i-1) \bmod N) + 1, ((i-1) \bmod M) + 1]$

Define the following functions:

$$\begin{aligned}
\text{rodriguez}(\overset{3}{\boldsymbol{r}}, \overset{3}{\boldsymbol{x}}) &= \underline{\text{let}}\ \theta = \|\boldsymbol{r}\|, \boldsymbol{v} = \boldsymbol{r}/\|\boldsymbol{r}\| \\
&\quad\ \underline{\text{in}}\ \boldsymbol{x}\cos\theta + (\boldsymbol{v} \times \boldsymbol{x})\sin\theta + \boldsymbol{v}(\boldsymbol{v} \cdot \boldsymbol{x})(1 - \cos\theta) \in \mathbb{R}^3 \\
\text{rodriguez}(\overset{3}{\boldsymbol{r}} = [0,0,0], \overset{3}{\boldsymbol{x}}) &= \boldsymbol{x} + (\boldsymbol{r} \times \boldsymbol{x}) \\
\text{p2e}(\overset{3}{\boldsymbol{x}}) &= [x_1/x_3, x_2/x_3] \in \mathbb{R}^2 \\
\text{distort}(\overset{2}{\boldsymbol{\kappa}}, \overset{2}{\boldsymbol{u}}) &= \boldsymbol{u}(1 + \kappa_1\|\boldsymbol{u}\|^2 + \kappa_2\|\boldsymbol{u}\|^4) \in \mathbb{R}^2 \\
\text{project}(\overset{3}{\boldsymbol{r}}, \overset{3}{\boldsymbol{c}}, \overset{1}{f}, \overset{2}{\boldsymbol{x}_0}, \overset{2}{\boldsymbol{\kappa}}, \overset{3}{\boldsymbol{x}}) &= \text{distort}(\boldsymbol{\kappa}, \text{p2e}(\text{rodriguez}(\boldsymbol{r}, \boldsymbol{x} - \boldsymbol{c})))f + \boldsymbol{x}_0 \in \mathbb{R}^2 \\
\text{reprojerr}(\overset{3}{\boldsymbol{r}}, \overset{3}{\boldsymbol{c}}, \overset{1}{f}, \overset{2}{\boldsymbol{x}_0}, \overset{2}{\boldsymbol{\kappa}}, \overset{3}{\boldsymbol{x}}, \overset{1}{w}, \overset{2}{\boldsymbol{m}}) &= w(\text{project}(\boldsymbol{r}, \boldsymbol{c}, f, \boldsymbol{x}_0, \boldsymbol{\kappa}, \boldsymbol{x}) - \boldsymbol{m}) \in \mathbb{R}^2 \\
\text{werr}(\overset{1}{w}) &= 1 - w^2 \in \mathbb{R}
\end{aligned}$$

(Note that 'reprojerr' is negated as compared to [51]; the existing implementations use the version as written here, which we copy so that we compare implementations of the same problem.)

Objective function (as function of $\boldsymbol{C}, \boldsymbol{X}, \boldsymbol{w}$):

$$\begin{aligned}
&\text{reproj_error} \in \mathbb{R}^{p \times 2}; \\
&\text{reproj_error}[i][..] = \underline{\text{let}}\ c = \boldsymbol{C}[obs[i][1]], \boldsymbol{x} = \boldsymbol{X}[obs[i][2]] \\
&\qquad\qquad\qquad\ \underline{\text{in}}\ \text{reprojerr}(c[1..3], c[4..6], c[7], c[8..9], c[10..11], \boldsymbol{x}, \boldsymbol{w}[i], \boldsymbol{F}[i]) \\
&\text{w_error} \in \mathbb{R}^p; \text{w_error}[i] = \text{werr}(\boldsymbol{w}[i])
\end{aligned}$$

Jacobian matrix:

$$J \in \mathbb{R}^{(2p+p)\times(11n+3m+p)};\ J[i][j] = \frac{\partial((\rightsquigarrow \text{reproj\_error}) +\!\!+ \text{w\_error})_i}{\partial((\rightsquigarrow \boldsymbol{C}) +\!\!+ (\rightsquigarrow \boldsymbol{X}) +\!\!+ \boldsymbol{w})_j}$$

where $(\rightsquigarrow M)$ means to flatten the matrix M in row-major order. For example, $(\rightsquigarrow \boldsymbol{C}) \in \mathbb{R}^{11n}$ contains n blocks of 11 scalars. This Jacobian is sparse, and to be represented in CSR format, where only entries that are statically known to be zero are omitted. Figure 4 shows a diagram of the sparsity pattern.

The task is to compute the Jacobian matrix J in CSR format.

C ADBench task implementations

C.1 GMM: Gaussian Mixture Model fitting

```haskell
{-# LANGUAGE DeriveGeneric, FlexibleContexts, PatternSynonyms, TypeOperators, ViewPatterns #-}

import Prelude ()
import Data.Array.Accelerate

type FLT = Float

-- First some data definitions. For each structure, we define the Haskell
-- datatype, and derive an instance of Arrays (or Elt for scalar-level
-- structures) to make them useable in the Accelerate EDSL. The pattern
-- synonyms are for convenience lifting/unlifting.

data GMMIn =
    GMMIn_ { gmmInAlphas   :: !(Vector FLT)  -- [k]
           , gmmInMeans    :: !(Matrix FLT)  -- [k][d]
           , gmmInICF      :: !(Matrix FLT)  -- [k][d + 1/2 d(d-1)]: d q's, then l's
           , gmmInX        :: !(Matrix FLT)  -- [n][d]
           , gmmInWisGamma :: !(Scalar FLT)
           , gmmInWisM     :: !(Scalar Int)
           }
  deriving (Generic)

deriving instance Show GMMIn
instance Arrays GMMIn

pattern GMMIn :: Acc (Vector FLT) -> Acc (Matrix FLT) -> Acc (Matrix FLT)
              -> Acc (Matrix FLT) -> Acc (Scalar FLT) -> Acc (Scalar Int) -> Acc GMMIn
pattern GMMIn a mu i x g m = Pattern (a, mu, i, x, g, m)
{-# COMPLETE GMMIn #-}

data GMMOut =
    GMMOut_ { gmmOutAlphas :: !(Vector FLT)  -- [k]
            , gmmOutMeans  :: !(Matrix FLT)  -- [k][d]
            , gmmOutICF    :: !(Matrix FLT)  -- [k][d + 1/2 d(d-1)]: d q's, then l's
            }
  deriving (Generic)

deriving instance Show GMMOut
instance Arrays GMMOut

pattern GMMOut :: Acc (Vector FLT) -> Acc (Matrix FLT) -> Acc (Matrix FLT) -> Acc GMMOut
pattern GMMOut a mu i = Pattern (a, mu, i)
{-# COMPLETE GMMOut #-}

-- This structure contains the two nontrivial precomputed values.
data Precomputed =
    Precomputed_ FLT  -- ^ 1/2 N D log(2 pi)
                 FLT  -- ^ K (n' D (log gamma - 1/2 log(2)) - multigammaln(1/2 n', D))
  deriving (Show, Generic)

instance Elt Precomputed

pattern Precomputed :: Exp FLT -> Exp FLT -> Exp Precomputed
pattern Precomputed c1 c2 = Pattern (c1, c2)
{-# COMPLETE Precomputed #-}

cAll :: Exp All
cAll = constant All

-- We elide the actual out-of-Accelerate precomputation.
precompute :: GMMIn -> Precomputed
precompute input = {- elided -}

logsumexp :: (Ord e, Floating e, Shape sh) => Acc (Array (sh :. Int) e) -> Acc (Array sh e)
logsumexp x =
    let len = indexHead (shape x)
        maxx = maximum x
        shiftedx = zipWith (-) x (replicate (lift Any ::. len) maxx)
    in zipWith (+) (map log (sum (map exp shiftedx))) maxx
```

```
70
71  sqsum :: (Num a, Shape sh) => Acc (Array (sh :. Int) a) -> Acc (Array sh a)
72  sqsum = sum . map (\x -> x * x)
73
74  -- The second term of L.
75  logWishartPrior :: Exp Int
76                  -> Acc (Matrix FLT) -> Acc (Vector FLT) -> Acc (Matrix FLT)
77                  -> Acc (Scalar FLT) -> Acc (Scalar Int)
78                  -> Exp Precomputed
79                  -> Acc (Scalar FLT)
80  logWishartPrior k qdiags sumQs lvals wisGamma wisM (Precomputed _ c2) =
81      let res1 = sum $
82              zipWith (-)
83                  (zipWith (*) (zipWith (+) (sqsum qdiags) (sqsum lvals))
84                               (replicate (I1 k) (map (\x -> 0.5 * x * x) wisGamma)))
85                  (zipWith (*) sumQs (replicate (I1 k) (map fromIntegral wisM)))
86      in map (subtract c2) res1
87
88  -- The full objective function. This does not yet include anything specific to
89  -- AD, except for the choice of expression in computing lmats. We just compute L.
90  objective :: Acc GMMIn -> Exp Precomputed -> Acc (Scalar FLT)
91  objective (GMMIn alphas means icf x wisGamma wisM) prec@(Precomputed c1 _) =
92      let I2 n d = shape x
93          I2 _ numQL = shape icf
94          numL = numQL - d  -- numL = 1/2 d (d - 1)
95          I1 k = shape alphas
96          qvals = backpermute (I2 k d) (\v -> v) icf
97          lvals = backpermute (I2 k numL) (\(I2 i j) -> I2 i (j + d)) icf
98          qdiags = map exp qvals
99          sumQs = sum qvals
100         shiftedX = zipWith (-) (replicate (I3 cAll k cAll) x)
101                                (replicate (I3 n cAll cAll) means)
102         -- If array indexing would be nice and efficient under AD, the 'generate'
103         -- definition of lmats would be the most readable. However, since it
104         -- isn't particularly, we will define it in terms of more information-
105         -- retaining primitives in the 'backpermute' definition of lmats.
106         lmats = if False  -- We do not use RebindableSyntax; this is a Haskell 'if'.
107             then generate (I4 n k d d)
108                           (\(I4 _ ki i1 i2) ->
109                               let li = d * (d - 1) `div` 2 - (d-1 - i2) * (d-1 - i2 + 1) `div` 2
110                                         + i1 - i2 - 1
111                               in cond (i1 > i2) (lvals ! I2 ki li) 0)
112             else let -- For lack of a backpermute-with-default primitive
113                      lvalsWithZero = zipWith (*)
114                                              (generate (I2 k (numL + 1)) (\(I2 _ j) -> cond (j == 0) 0 1))
115                                              (backpermute (I2 k (numL + 1)) (\(I2 i j) -> I2 i (j + d - 1)) icf)
116                  in backpermute (I4 n k d d)
117                                 (\(I4 _ ki i1 i2) ->
118                                     let li = d * (d - 1) `div` 2 - (d-1 - i2) * (d-1 - i2 + 1) `div` 2
119                                               + i1 - i2 - 1
120                                     in cond (i1 > i2) (I2 ki (li + 1)) (I2 ki 0))
121                                 lvalsWithZero
122         eqxprod = zipWith (*) (replicate (I3 n cAll cAll) qdiags) shiftedX
123         lxprod = sum (zipWith (*) lmats (replicate (I4 cAll cAll d cAll) shiftedX))
124         innerTerm = zipWith (-) (replicate (I2 n cAll) (zipWith (+) alphas sumQs))
125                                 (map (* 0.5) $ sqsum (zipWith (+) eqxprod lxprod))
126         slse = sum (logsumexp innerTerm)
127     in zipWith (\a1 a2 -> -c1 + a1 + a2)
128                (zipWith (-) slse
129                             (map (fromIntegral n *) (logsumexp alphas)))
130                (logWishartPrior k qdiags sumQs lvals wisGamma wisM prec)
131
132 -- Compute the gradient using AD. 'gradientA' is the new Accelerate primitive
133 -- that our reverse AD implementation adds to the language. Here it computes
134 -- the gradient with respect to _all_ input arguments, but we return only those
135 -- that we actually need (recall from the definition of GMMIn that Q and L are
136 -- together in a single array, ICF).
137 jacobian :: Precomputed -> Acc GMMIn -> Acc GMMOut
138 jacobian prec input =
139     let GMMIn a m i _ _ _ = gradientA (\input' -> objective input' (constant prec)) input
140     in GMMOut a m i
```

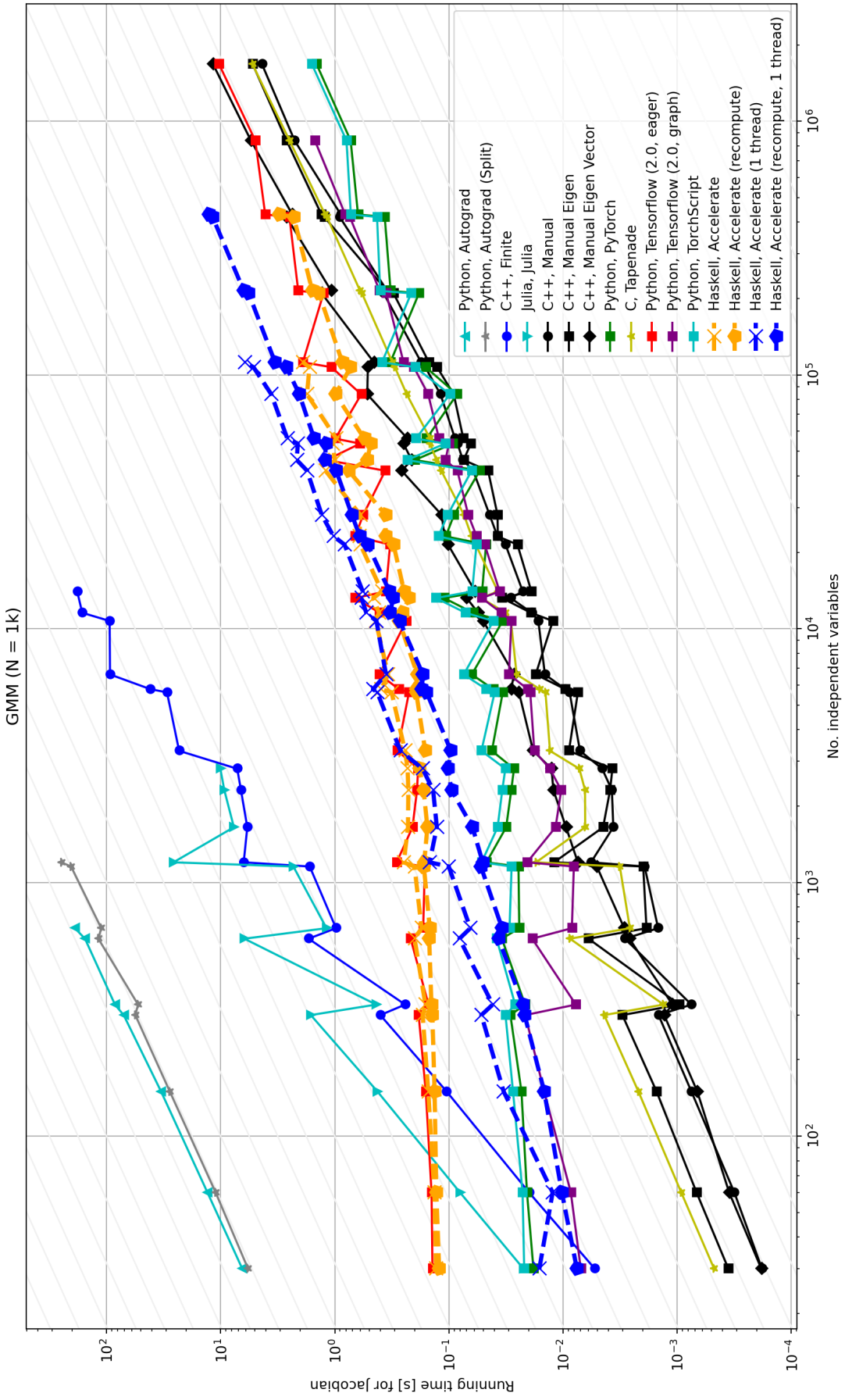
C.2 BA: Bundle Adjustment

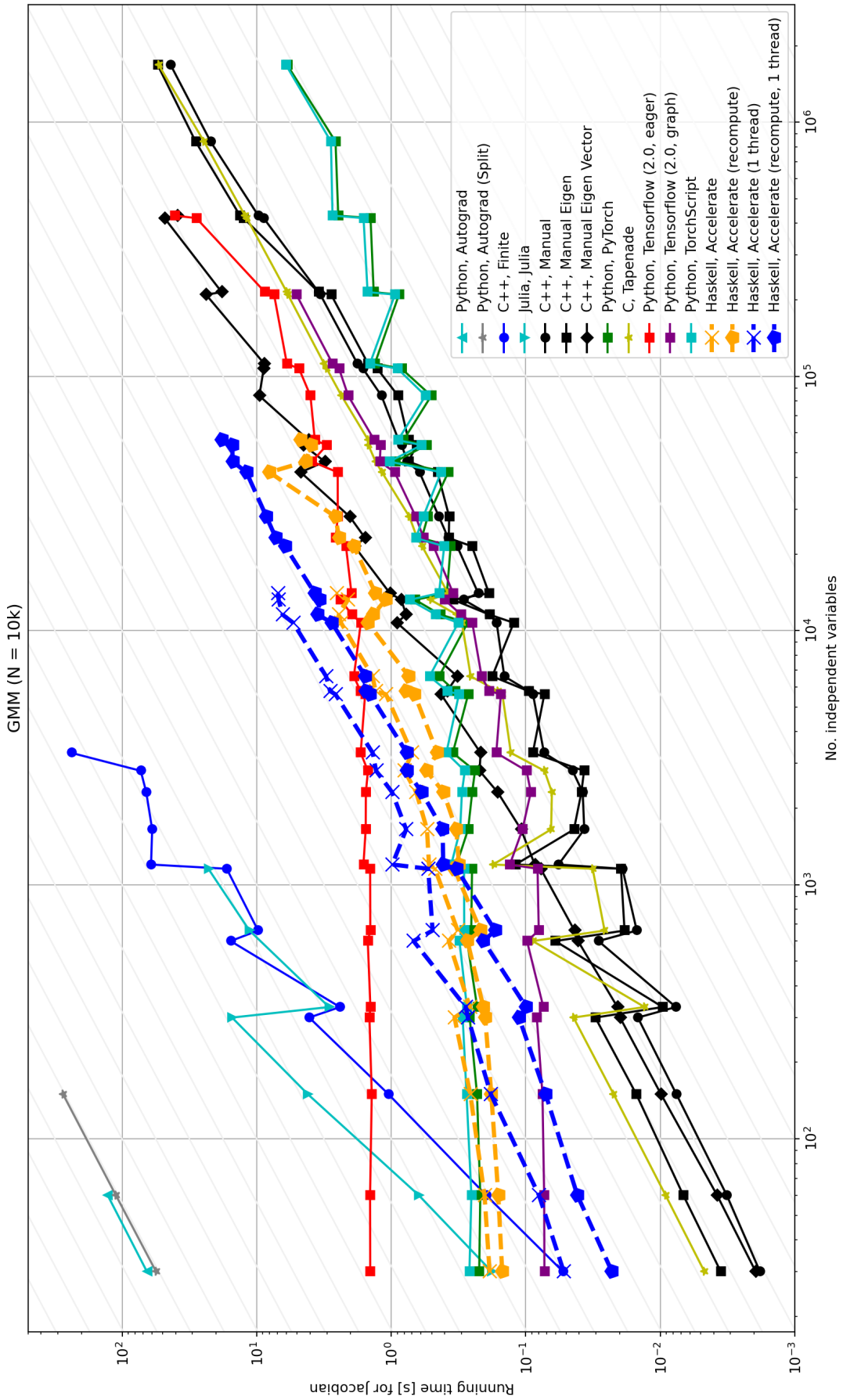
```haskell
{-# LANGUAGE DeriveGeneric, FlexibleContexts, FlexibleInstances, MultiParamTypeClasses, TypeApplications #-}

import Prelude ()
import Data.Array.Accelerate hiding (pi)

type FLT = Float
type Pt3D = (FLT, FLT, FLT)
type Pt2D = (FLT, FLT)

data Camera =
    Camera_ { camR     :: !Pt3D
            , camC     :: !Pt3D
            , camF     :: !FLT
            , camX0    :: !Pt2D
            , camKappa :: !Pt2D }
  deriving (Generic)

deriving instance Show Camera
instance Elt Camera

pattern Camera :: Exp Pt3D -> Exp Pt3D -> Exp FLT -> Exp Pt2D -> Exp Pt2D -> Exp Camera
pattern Camera r c f x0 k = Pattern (r, c, f, x0, k)
{-# COMPLETE Camera #-}

data Observation =
    Observation_ { obsCamIdx :: !Int
                 , obsPtIdx  :: !Int }
  deriving (Generic)

deriving instance Show Observation
instance Elt Observation

pattern Observation :: Exp Int -> Exp Int -> Exp Observation
pattern Observation ci pi = Pattern (ci, pi)
{-# COMPLETE Observation #-}

data BAIn =
    BAIn_ { baInCams  :: !(Vector Camera)        -- [n]
          , baInPts   :: !(Vector Pt3D)          -- [m]
          , baInW     :: !(Vector FLT)           -- [p]
          , baInFeats :: !(Vector Pt2D)          -- [p]
          , baInObs   :: !(Vector Observation) } -- [p]
  deriving (Generic)

deriving instance Show BAIn
instance Arrays BAIn

pattern BAIn :: Acc (Vector Camera) -> Acc (Vector Pt3D) -> Acc (Vector FLT)
             -> Acc (Vector Pt2D) -> Acc (Vector Observation) -> Acc BAIn
pattern BAIn c p w f o = Pattern (c, p, w, f, o)
{-# COMPLETE BAIn #-}

data BAOut =
    BAOut_ { baOutRows :: !(Vector Int)   -- [2p + p]
           , baOutCols :: !(Vector Int)   -- [11n + 3m + p]
           , baOutVals :: !(Vector FLT) } -- [31p]
  deriving (Generic)

deriving instance Show BAOut
instance Arrays BAOut

pattern BAOut :: Acc (Vector Int) -> Acc (Vector Int) -> Acc (Vector FLT) -> Acc BAOut
pattern BAOut r c v = Pattern (r, c, v)
{-# COMPLETE BAOut #-}

-- Pick a parameter from a Camera definition based on its index. Note that the
-- list here is on the meta-level from the perspective of Accelerate;
-- Accelerate, and later LLVM, only see a number of nested conditionals. This
-- function is never differentiated.
camIndex :: Exp Camera -> Exp Int -> Exp FLT
camIndex (Camera (T3 r1 r2 r3) (T3 c1 c2 c3) f (T2 x1 x2) (T2 k1 k2)) =
    pick [r1, r2, r3, c1, c2, c3, f, x1, x2, k1, k2]
```

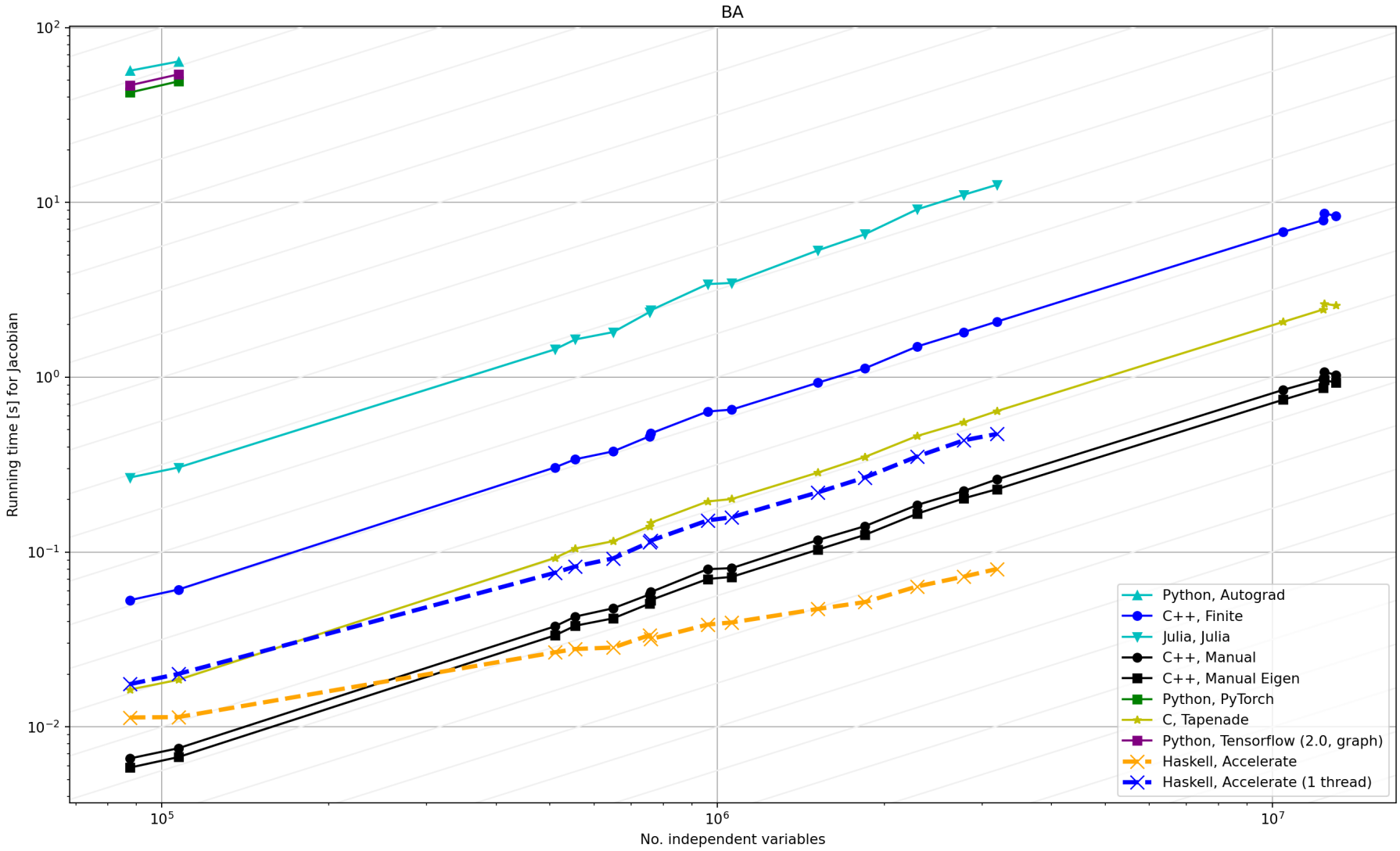
```
73      where
74        pick [x] _ = x
75        pick (x:xs) i = cond (i == 0) x (pick xs (i-1))
76        pick [] _ = error "pick: what"
77
78  ptIndex :: Exp Pt3D -> Exp Int -> Exp FLT
79  ptIndex (T3 x y z) i = cond (i == 0) x (cond (i == 1) y z)
80
81
82  vecadd :: TLift t FLT => Exp t -> Exp t -> Exp t
83  vecadd = tlift @_ @FLT (+)
84
85  vecsub :: TLift t FLT => Exp t -> Exp t -> Exp t
86  vecsub = tlift @_ @FLT (-)
87
88  dot :: TLift t FLT => Exp t -> Exp t -> Exp FLT
89  dot v w = treduce (+) (tlift @_ @FLT (*) v w)
90
91  scale :: TLift t FLT => Exp FLT -> Exp t -> Exp t
92  scale = tlift1 (*)
93
94  cross :: Exp Pt3D -> Exp Pt3D -> Exp Pt3D
95  cross (T3 a b c) (T3 x y z) = T3 (b*z - c*y) (c*x - a*z) (a*y - b*x)
96
97  rodriguez :: Exp Pt3D -> Exp Pt3D -> Exp Pt3D
98  rodriguez r@(T3 r1 r2 r3) x =
99      let sqtheta = r1 * r1 + r2 * r2 + r3 * r3
100         theta = sqrt sqtheta
101         v = T3 (r1 / theta) (r2 / theta) (r3 / theta)
102     in cond (sqtheta == 0)
103             (x `vecadd` cross r x)
104             (scale (cos theta) x
105                 `vecadd` scale (sin theta) (cross v x)
106                 `vecadd` scale ((v `dot` x) * (1 - cos theta)) v)
107
108 p2e :: Exp Pt3D -> Exp Pt2D
109 p2e (T3 x1 x2 x3) = T2 (x1 / x3) (x2 / x3)
110
111 distort :: Exp Pt2D -> Exp Pt2D -> Exp Pt2D
112 distort (T2 k1 k2) u@(T2 u1 u2) =
113     let lensq = u1 * u1 + u2 * u2
114     in scale (1 + k1 * lensq + k2 * lensq * lensq) u
115
116 project :: Exp Camera -> Exp Pt3D -> Exp Pt2D
117 project (Camera r c f x0 kappa) x =
118     scale f (distort kappa (p2e (rodriguez r (x `vecsub` c)))) `vecadd` x0
119
120 reprojerr :: Exp Camera -> Exp Pt3D -> Exp FLT -> Exp Pt2D -> Exp Pt2D
121 reprojerr cam x w m = scale w (project cam x `vecsub` m)
122
123 werr :: Exp FLT -> Exp FLT
124 werr w = 1 - w * w
125
126 -- We compute the Jacobian chunks in the following packed arrays:
127 --   - 2p x 11: d(flatten(reproj_error)) / d(cams_{obs[i]})
128 --   - 2p x 3: d(flatten(reproj_error)) / d(pts_{obs[i]})
129 --   - 2p: d(flatten(reproj_error)) / d(w_i)
130 --   - p: d(w_error) / d(w_i)
131 jacobianPacked :: Acc BAIn -> Acc (Matrix FLT, Matrix FLT, Vector FLT, Vector FLT)
132 jacobianPacked (BAIn cams pts ws feats obs) =
133     let I1 p = shape obs
134         grads :: Acc (Vector ((Camera, Camera), (Pt3D, Pt3D), (FLT, FLT)))
135         grads = map (\(T2 i (Observation ci pi)) ->
136                         let T4 dcam1 dpt1 dw1 _ =
137                                 gradientE (\(T4 cam pt w feat) -> fst (reprojerr cam pt w feat))
138                                           (T4 (cams ! I1 ci) (pts ! I1 pi) (ws ! i) (feats ! i))
139                             T4 dcam2 dpt2 dw2 _ =
140                                 gradientE (\(T4 cam pt w feat) -> snd (reprojerr cam pt w feat))
141                                           (T4 (cams ! I1 ci) (pts ! I1 pi) (ws ! i) (feats ! i))
142                         in T3 (T2 dcam1 dcam2) (T2 dpt1 dpt2) (T2 dw1 dw2))
143                     (indexed obs)
144         campacked = generate (I2 (2 * p) 11)
145                              (\(I2 i j) -> let T3 (T2 dcam1 dcam2) _ _ = grads ! I1 (i `div` 2)
```

```
                                        in camIndex (cond (i `mod` 2 == 0) dcam1 dcam2) j)
        ptpacked = generate (I2 (2 * p) 3)
                            (\(I2 i j) -> let T3 _ (T2 dpt1 dpt2) _ = grads ! I1 (i `div` 2)
                                          in ptIndex (cond (i `mod` 2 == 0) dpt1 dpt2) j)
        rwpacked = generate (I1 (2 * p))
                            (\(I1 i) -> let T3 _ _ (T2 dw1 dw2) = grads ! I1 (i `div` 2)
                                        in cond (i `mod` 2 == 0) dw1 dw2)
        wwpacked = map (gradientE werr) ws
    in T4 campacked ptpacked rwpacked wwpacked

jacobian :: Acc BAIn -> Acc BAOut
jacobian input@(BAIn cams pts _ _ obs) =
    let I1 n = shape cams
        I1 m = shape pts
        I1 p = shape obs

        T4 dcams dpts drdw dwdw = jacobianPacked input

        rows = enumFromStepN (I1 (2 * p)) 0 15 ++ enumFromN (I1 (p + 1)) (2 * p * 15)
        colsvals =
            generate (I1 (2 * p * 15))
                     (\(I1 i) -> let obsi = i `div` (2 * 15)
                                     valouti = i `div` 15
                                     Observation ci pi = obs ! I1 obsi
                                     j = i `mod` 15
                                 in cond (j < 11)
                                         (T2 (11 * ci + j) (dcams ! I2 valouti j))
                                         (cond (j < 14)
                                               (T2 (11 * n + 3 * pi + j - 11) (dpts ! I2 valouti (j - 11)))
                                               (T2 (11 * n + 3 * m + obsi) (drdw ! I1 valouti))))
            ++
            generate (I1 p)
                     (\(I1 i) -> T2 (11 * n + 3 * m + i) (dwdw ! I1 i))
    in BAOut rows (map fst colsvals) (map snd colsvals)


class (Elt t, Elt a) => TLift t a where
    tlift :: (Exp a -> Exp a -> Exp a) -> Exp t -> Exp t -> Exp t
    tlift1 :: (Exp a -> Exp a -> Exp a) -> Exp a -> Exp t -> Exp t
    treduce :: (Exp a -> Exp a -> Exp a) -> Exp t -> Exp a

instance Elt a => TLift a a where
    tlift f = f
    tlift1 f = f
    treduce _ x = x

instance (TLift t1 a, TLift t2 a) => TLift (t1, t2) a where
    tlift f (T2 a1 a2) (T2 b1 b2) = T2 (tlift f a1 b1) (tlift f a2 b2)
    tlift1 f a (T2 b1 b2) = T2 (tlift1 f a b1) (tlift1 f a b2)
    treduce f (T2 x y) = treduce f x `f` treduce f y

instance (TLift t1 a, TLift t2 a, TLift t3 a) => TLift (t1, t2, t3) a where
    tlift f (T3 a1 a2 a3) (T3 b1 b2 b3) = T3 (tlift f a1 b1) (tlift f a2 b2) (tlift f a3 b3)
    tlift1 f a (T3 b1 b2 b3) = T3 (tlift1 f a b1) (tlift1 f a b2) (tlift1 f a b3)
    treduce f (T3 x y z) = treduce f x `f` treduce f y `f` treduce f z
```

D Benchmark graphs




GMM (N = 10k)
Running time [s] for Jacobian
No. independent variables
Python, Autograd
Python, Autograd (Split)
C++, Finite
Julia, Julia
C++, Manual
C++, Manual Eigen
C++, Manual Eigen Vector
Python, PyTorch
C, Tapenade
Python, Tensorflow (2.0, eager)
Python, Tensorflow (2.0, graph)
Python, TorchScript
Haskell, Accelerate
Haskell, Accelerate (recompute)
Haskell, Accelerate (1 thread)
Haskell, Accelerate (recompute, 1 thread)


BA
Running time [s] for Jacobian
No. independent variables
10^2
10^1
10^0
10^{-1}
10^{-2}
10^5
10^6
10^7
Python, Autograd
C++, Finite
Julia, Julia
C++, Manual
C++, Manual Eigen
Python, PyTorch
C, Tapenade
Python, Tensorflow (2.0, graph)
Haskell, Accelerate
Haskell, Accelerate (1 thread)