Maintaining parallelism in reverse-mode automatic differentiation on functional parallel array languages

Simon van Hus
6147879
s.vanhus@students.uu.nl

Abstract

In this paper we set out to make a simple reverse-mode automatic differentiation (AD) algorithm, that uses tracing for the forward pass, and preserves data parallelism in the reverse pass. To do this, we first try to formalize the notion of tracing somewhat. We find that while some flexibility in the definition of is needed for it to work well, we can also boil it down to picking a subset of data types to keep in the trace. We also define a couple of logical assertions that further help us in showing whether a trace does really contain the information that we need. Having defined tracing, in theory, but also over a Haskell DSL, we continue to automatic differentiation. Here we expand the tracing function into a forward-pass function by adding reference counting and intermediate values. Using this forward-pass trace as a map, we then show how we can do the reverse-pass. We also show that we can keep data-parallelism intact for the map and fold (reduce) operations. Finally, we also highlight how task parallelism can be used in the reverse-pass to possibly unlock even more efficiency.

1 Introduction

In Automatic Differentiation (AD) we try to find the derivative of some numeric function, automatically. AD systems are nothing new, however they are still as relevant as ever. Especially with the current AI spring, we find AD playing the important role as the back propagator function in artificial neural networks.

And while AD is fairly quick, it also has its inefficiencies. Especially in array languages, we find that modern AD libraries either do not really take advantage of data parallelism, or do so in convoluted ways.

In this paper, we use the intuitive method of finding the computational graph of a function through tracing. Then we do a reverse-pass over this computational graph to calculate the derivative. By taking parallel array operations into careful consideration, we can even maintain data parallelism by using other data-parallel operations to calculate the partial derivatives.

We set out to think a little more sceptically about what tracing actually means, and

then use our new understanding for finding the derivative using reverse-mode AD.

2 Background

2.1 Automatic Differentiation

Automatic Differentiation (AD), like the name suggests, involves programmatically finding the derivative of some programmed function [1]. The other main method for programmatically finding the derivative of a function is numerical differentiation, which uses the finite difference method. By adjusting the input(s) to the function by a very small number, we can see the effect on the output(s) of the function. Unfortunately, due to the way real numbers are represented using floating-point computation, this method is prone to round-off error (or truncation error). AD avoids this by actually performing the differentiation on a program, to produce the differentiated program. This is very similar to how a human would differentiate a mathematical function (sometimes called symbolic or manual differentiation), but performed on a computer program.

AD makes very explicit use of the chain rule of partial derivatives of compound functions, which provides a method for finding the derivative of compound functions and states that we can combine partial derivatives of parts of the function together into the complete derivative. Say we have some single-variate function $h(x)$, which is the compound function of the functions f and g:

$$h(x) = (f \circ g)(x)$$

In this case, the chain rule tells us that the derivative of $h(x)$ is given by as $\frac{d}{dx}h(x)$:

$$\frac{d}{dx}h(x) = \frac{d}{dx}(f \circ g)(x) = \frac{d(f(g(x)))}{dg} \cdot \frac{d(g(x))}{dx}$$

For clarity, in Lagrange's notation, where $h'(x)$ is the derivative of $h(x)$, this same statement can be expressed as:

$$h'(x) = (f \circ g)'(x) = f'(g(x)) \cdot g'(x)$$

The chain rule also extends to compositions of more than two functions. For example, say we have a function $k(x)$ as below:

$$k(x) = (f \circ g \circ h)(x)$$

We can then find the derivative of $k(x)$ using the chain rule as well:

$$\frac{d}{dx}k(x) = \frac{d}{dx}(f \circ g \circ h)(x) = \frac{d(f(g(h(x))))}{dg} \cdot \frac{d(g(h(x)))}{dh} \cdot \frac{d(h(x))}{dx}$$

Again for clarity, in Lagrange's notation this would be:

$$\begin{aligned} k'(x) = (f \circ g \circ h)'(x) &= f'((g \circ h)(x)) \cdot (g \circ h)'(x) \\ &= f'((g \circ h)(x)) \cdot g'(h(x)) \cdot h'(x) \end{aligned}$$

The chain rule also provides us with a method of deriving multivariate functions. For instance, we can imagine a function $f(x, y)$. Now, the derivative of f changes depending

on which variable we wish to derive with respect to. Furthermore, this is not a composition of functions, so the chain rule does not come into play. However, if we image the variables x and y as single-variable functions $x(t)$ and $y(t)$ we can find the derivative of f with respect to t using the chain rule. We get:

$$f(x(t), y(t))$$

Now to calculate the derivate of f with respect to t, we first need to find the derivative of x with respect to t and the derivative of y with respect to t. The chain rule tells us that the derivative of f here is equal to the partial derivative of f with respect to x summed with the partial derivative of f with respect to y. We can express this as:

$$\frac{d}{dt}f(x(t), y(t)) = \frac{\partial(f(x(t), y(t)))}{\partial x} \cdot \frac{d(x(t))}{dt} + \frac{\partial f(x(t), y(t))}{\partial y} \cdot \frac{d(y(t))}{dt}$$

An important thing to note about the chain rule is that we still need the intermediate primal values in a compound function. Review the following compound function:

$$(f \circ g \circ h \circ k)(x)$$

In Lagrange's notation, the derivative becomes:

$$\begin{aligned}(f \circ g \circ h \circ k)'(x) &= f'((g \circ h \circ k)(x)) \cdot g'((h \circ k)(x)) \cdot h'(k(x)) \cdot k'(x) \\ &= f'(g(h(k(x)))) \cdot g'(h(k(x))) \cdot h'(k(x)) \cdot k'(x)\end{aligned}$$

See how on the second line we have highlighted the primal parts of the equation, the intermediate values that we need for finding the derivative. Also note how values deeper in the chain are used multiple times; $h(k(x))$ is used twice: first in the derivative f' and second in the derivative g'. $k(x)$ is even used three times. Looking at this example, it becomes very clear that it would be more efficient to calculate $k(x)$ once and save that result somehow, rather than recalculating it every time it came up. The storing and reusing of intermediate values is a fundamental property of AD, and is called "sharing".

To actually implement automatic differentiation, we seek to break the target program down to its most basic mathematical operations, for which we know the derivatives. Then we can use the chain rule to combine them together into the derivative of the whole program. There are two main ways to actually resolve these derivatives: using either forward accumulation or backward/reverse accumulation. When applied in AD implementations these are commonly respectively referred to as forward-mode and reverse-mode. Both methods are described in the 1986 paper "The arithmetic of differentiation" by B. Rall [2].

In forward-mode AD we move through the program to differentiate in normal execution order. By knowing which input variable we wish to differentiate, we can compute every step of the derivative as our inputs are used by the program. Rall demonstrates this using a method known as dual-numbers, where each real number is represented by a pair of numbers, similar to complex numbers. In dual-numbers, the first number in the pair represents the primal part of the number, whereas the second number represents the derivative part (called the tangent in forward-mode). When we compute with these numbers through arithmetic operations, we can operate on the primal parts as normal, and use derivative rules to calculate the derivative of the result using the tangent parts. An example of this is given in Equation 1, where $\dot{a}$ is the tangent part of some real number a.

$$(a, \dot{a}) \cdot (b, \dot{b}) = (a \cdot b, \dot{a} \cdot b + \dot{b} \cdot a) \tag{1}$$

Now we can find the derivative of some program with regard to the input x_i by setting $\dot{x_i}$ to 1, setting the tangents of all other inputs to 0, and just running through the program calculating tangents as we go. The tangent part of the output value(s) is also the calculated derivative of the whole program.

While forward-mode AD is fairly straightforward, it comes with some drawbacks. The main one being that for a function $f : \mathbb{R}^n \to \mathbb{R}^m$ with n inputs and m outputs, to get the effect of each input variable on each output variable, we would need to perform n passes over the function, one for each input variable (or we need to track n tangent parts for each step). This can be cumbersome, especially if n is much larger than m. For those cases, we might be better off with reverse accumulation, or reverse-mode.

In reverse-mode, we peg the derivative part of one of our outputs with some seed (often 1), and set the derivative parts of the other outputs to 0. These derivative parts are generally referred to as adjoints instead of tangents in reverse-mode. When the outputs are set, we can work our way back through the function, calculating the derivative parts from the output to the input. Intuitively, this computes the gradient of the output dimension we pegged to 1, or the direction of the steepest slope. Practically, the idea of working back through a program requires some way of knowing where the outputs came from (a sort of dependency structure). This then requires a forward pass, to find this structure, to calculate the intermediate values, and often to set up any dual-numbers or other implementation details. And while reverse-mode is definitely harder to implement, it also provides us with a way to calculate the sensitivity of all inputs to an output, which is much more efficient for functions with many more inputs than outputs (which can be quite common in certain applications like neural networks).

In mathematical terms, calculating the partial derivative of one output with regard to one input, means calculating one cell in the Jacobian, the matrix of all partial derivatives. For a function $f : \mathbb{R}^n \to \mathbb{R}^m$ with n inputs and m outputs, the Jacobian J_f would be an $n \times m$ matrix. Here a column i represents the partial derivatives $\frac{\partial \vec{f}}{\partial x_i}$, where $\vec{f}$ are all outputs of f, and x_i represents a single input. A row j then represents the derivatives $\nabla f_j = \frac{\partial f_j}{\partial \vec{x}}$, where $\vec{x}$ are all inputs of f, and ∇f_j is also known as the gradient of the single output value f_j. This is also shown in Equation 2, showing the Jacobian for some function f with n inputs $(x_1, \ldots, x_n)$ and m inputs $(f_1, \ldots, f_m)$. An important take-away here is that forward-mode computes the derivatives of all outputs with regard to a single input, so a column in the Jacobian, and reverse-mode computes the derivatives of all inputs with regard to a single output, so a row in the Jacobian. Again, if we want to calculate the full Jacobian, forward-mode is more efficient when we have more outputs than inputs or when the Jacobian has more columns than rows, and the reverse-mode is more efficient for functions with more inputs than outputs or for Jacobians with more rows than columns.

$$J_f = \begin{bmatrix} \frac{\partial \vec{f}}{\partial x_1}, \ldots, \frac{\partial \vec{f}}{\partial x_n} \end{bmatrix} = \begin{bmatrix} \nabla f_1 \\ \vdots \\ \nabla f_m \end{bmatrix} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix} \tag{2}$$

While it has long been known that reverse-mode automatic differentiation could be executed in time equal to some constant multiple of the execution time of the primal program [3], it seemed that a constant multiple of the execution memory was also needed, which could become very expensive for large programs. However, in 1992, Andreas Griewank showed that by using taping and checkpointing we could trace time complex-

ity for space complexity to reduce either to a constant multiple of the log of the execution time [4]. In general the practice of taping refers to a form of tracing on the program we wish to differentiate, where we execute the program as normal and record all the steps and intermediate values in a first-in-last-out data structure referred to as a "tape" or Wengert list. In a second phase to the reverse-mode algorithm, the tape is then used to calculate the derivatives in question, which due to the first-in-last-out nature of the tape, is in the precise reverse of the execution order of the program. An important advantage of taping is that by giving each variable and intermediate calculation a unique ID we can avoid redundant execution, because we can just refer to the intermediate value or tangent/adjoint stored in the tape. While taping is efficient time-wise, it clearly adds a memory overhead that can be quite sizable for large programs. Checkpointing aims to address this by storing multiple parts of the tape to memory attached to checkpoints in the program's execution. The trick here being, that on the reverse-pass only the intermediate values from the most recently encountered checkpoint are loaded from memory, intermediate values that were not stored as part of this checkpoint are recalculated. By strategically placing these checkpoints, and deciding which intermediate values are stored, this can cut the size complexity at a relatively small-time complexity increase. It should be noted that automatic differentiation can also be performed on a program where we do not have any specific inputs. We can do this using source transformation [5]. In its most basic form source transformation can be implemented as just interlacing the derivative calculations into the regular program. An example of this is provided in Listing 1, where we calculate the derivative of some variable y (as dy) with regard to the variable x1. For reverse-mode AD this kind of interlacing is not possible, as we need to reach the end of the program before we can start the reverse pass, which is exactly why we record our steps on the tape: so we can reverse over the tape and know how to produce our reverse AD program. An example of this is provided in Listing 2, where we again calculate the derivative of y (as dy), with regard to x1 and x2. So, to summarize, for a function $f : A \to B$, source transformation finds the derivative function for any input in the domain A, whereas dual-numbers (or similar approaches) find the derivative function for a specific input $a \in A$. Of course, in complex functions with a lot of control flow, source transformation can become cumbersome as it needs to account for all possible inputs, whereas dual-numbers only needs to account for one.

```
 x1 = 15
dx1 = 1
 x2 = 7
dx2 = 0
 r1 = x1 + x2
dr1 = dx1 + dx2
 y  = r1 × x2
dy  = r1 × dx2 + dr1 × x2
```

Listing 1: An example of forward mode AD by source transformation, with the AD statements in red

For forward-mode AD, the evaluation of the derivative is done during execution. Like in 1996's FADBAD package, which provided both forward-mode and reverse-mode AD for C++ [6]. The reverse-mode uses the taping method described by Griewank, implemented through a method called operator overloading. In forward-mode, operator overloading refers to providing the basic mathematical operators with methods that work on the numbers represented by a pair (of a primal part and a tangent part); this is the dual-numbers approach we mentioned before. For reverse-mode, operator overload-

```
 x1 = 15
 x2 = 7
 r1 = x1 + x2
 y  = r1 × x2

dy  = 1
dr1 = dy × x2
dx2 = dy × r1 + dr1 × 1
dx1 = dr1 × 1
```

Listing 2: An example of reverse mode AD by source transformation, with the AD statements in red

ing is used to rewrite the basic mathematical operators, so they record their use and intermediate values to a single tape data structure. A similar implementation was also provided by Griewank et al. in the 1996 package ADOL-C [7], again in 2001 using more efficient expression templates by Aubert et al. [8], and later in 2014 by Robin Hogan [9].

Source-code transformation is eventually also implemented, in the Tapenade AD program [10]. Tapenade adds derivative calculations to the code, but also employs lazy/delayed evaluation in the forward pass. This allows Tapenade to do some activity analysis, which in turn allows it to combine or discard some partial derivatives to be more efficient. It also implements the previously discussed checkpointing, where part of the tape is stored to be restored and differentiated later. This, in theory, allows for differentiating programs of arbitrary size, because the differentiation process is not limited by the size of the working memory [11].

More recently implementations, like Fei Wang et al. 2019 paper, have shown how to simplify reverse automatic differentiation using continuation passing style and delimited continuations [12]. This method uses dual numbers and cleverly overloads operators, so they call the forward pass as a continuation and then perform the backwards pass on the returned value.

In 2022, Krawiec et al. show how reverse-mode AD can be extended efficiently to higher-order functional programs [13]. While the Wang paper also did this, Krawiec uses the functional nature to provide a correctness proof of the reverse-mode AD, something that had previously only been done on implementations that were either asymptotically inefficient or only worked on first-order languages. They do however need taping again to make it provable and efficient.
Vákár and Smeding provide a provably correct form of higher-order reverse AD without taping in their 2022 paper [14], based on earlier work by Elliott in 2018 [15].

And in 2022 as well, Schenck et al. show how to do both forward-mode and reverse-mode automatic differentiation on second-order array language with nested data parallelism [16]. They do this by eliminating taping again, which forces sequential execution, by allowing potential redundant execution. But by limiting their AD implementation to second-order functional languages, they can largely avoid this redundancy with efficient program transformations on parallel operators.

Finally, in 2023, Smeding en Vákár bring back explicit dual-numbers to reverse AD [17]. However, instead of pairing each number with its computed adjoint, they instead pair it

with a linear back-propagator function, which they can then later chain to get the full derivative. While this initially seems to eliminate the need for taping, they find that through optimizations they return to a concept that is very close to taping and show that it is in fact equivalent.

2.2 Tracing

Tracing is a concept in computer science that is often left without proper definition. While the main ideas behind tracing are well known, they are generally assumed known by the reader and therefore left without explanation. This is also in part because, in software engineering, the term tracing also refers to finding the origin of some call ("tracing" the call stack), which is only tangentially related to the tracing we are interested in, but can leave definitions of tracing a bit muddled. This is why, in Section 3, we will discuss more about that proper definition. For now, it is important to know that, when we refer to tracing in this paper, we speak about tracing the path of computation through a program, given some (valid) input to said program. In other terms, given a program and an input, we walk through the program and record each computational step for some later purpose, like automatic differentiation. This recording can happen with some domain-specific pseudo-language, or in full-fledged code if we wish to reevaluate the trace later (or a combination of the two).

Doing tracing gives us some interesting insights into a program we trace. First, it effectively ignores control flow. This is fairly intuitive, when given a set of inputs to a program, the control flow will control what path the program uses, and since we only record computations we find what is often dubbed a "straight-line program" for some inputs. This can be useful for instance in automatic differentiation, where we often only want to differentiate a computation, not the entire program including unused branches. In this paper, we will also be using tracing for this purpose, as laid out in Section 4.

As mentioned, in literature we see this type of tracing used for automatic differentiation. One of these uses was by Bischof in 1991 [18]. In his paper, Bischof discusses the use of the computational graph of a program in automatic differentiation (using ADOL-C [7]). The computational graph of a program is a directed acyclic graph, where each node contains a computational step in the program, and edges connect these steps in the execution order of the program. Bischof creates this graph from the tape produced by ADOL-C, which makes sense: for automatic differentiation as discussed, the tape acts as a sort of trace, recording the steps that are important in the automatic differentiation. Bischof then uses a graph colouring algorithm on the computational graph to highlight "component functions" that may be differentiated concurrently, as to improve the running time of the algorithm. In 2008, Bischof et al. expand on this by extending the tracing automatic differentiation to loops [19]. They do this by extending ADOL-C, paying specific attention the parallelization opportunities present in automatic differentiation.

In a similar vein, Dougal Maclaurin presented in his PhD thesis in 2016 [20] a paper introducing Autograd. A software package to automatically differentiate Python code (including AD for the vector library Numpy). As Python is an expressive JIT-compiled (Just In Time) dynamic language, they opt for tracing to construct the computing graph on the fly when a function is called, and like Bischof's work this allows them to do the backwards pass off reverse-mode AD on the computational graph. They do this by wrapping their variables as "nodes" in the computational graph. When a variable is

used, it is first unwrapped for use, and then the result of whatever operation used the variable, is stored as a new variable and wrapped as well. The original variable and the produced variable are then linked such that the produced variable stores a reference to the original variable. This creates the reverse computational graph, which is exactly what is needed for the reverse AD pass.

TensorFlow, a machine learning library, also uses tracing to create computational graphs [21]. This kind of tracing is not as low to the ground as actually following individual computations. Since TensorFlow mainly focuses on building artificial neural networks, the computational graph is made explicit by the programmer. While there are some nuance differences between a computational graph of a neural network and the neural network itself, these differences are somewhat unimportant. More interestingly, TensorFlow allows for the partial execution of the computational graph. While Bischof's use of a graph colouring algorithm already suggested this, TensorFlow actively uses this technique to re-run partial computational graphs, which works well for the explicit nature of neural networks, as the computational graph stays unchanged even if the inputs change (as neural networks do not have internal control flow).

Finally, 2018's JAX uses tracing to enhance performance of general machine learning code [22]. The programmer annotates functions to be analysed by JAX, which then traces as optimizes them. Rather than finding the computational graph (or predefining it), JAX waits for Python to execute the function and actually traces it. Then, JAX optimizes it, mainly through a process called fusion, which is discussed in Section 2.3. This is also where JAX gets its name: Just After eXecution, as it waits for Python to execute the function first. It should be noted that JAX can only do this for functions which are pure-and-statically-composed (PSC), meaning functions that have no side effects and that do not change with different inputs. Again, machine learning code is especially suited for this, as it often already satisfies this PSC assumption.

2.3 Functional Parallel Array Programming

Array programming languages are programming languages that treat the array as a central data structure. This generally includes that functions, both user-defined and built-in, could be applied to arrays through vectorization. Vectorization involves applying a function to every element of an array at the same time. For instance, vectorization of addition would add two arrays together element-wise. This is shown in Equation 3, where $\vec{a}$ and $\vec{b}$ both are arrays of the same size.

$$\vec{a} + \vec{b} = [a_1 + b_1, \ldots, a_n + b_n] \text{ iff } |\vec{a}| = |\vec{b}| \tag{3}$$

In general vectorization would only work for arrays of the same size were it not for another central concept: broadcasting. Broadcasting involves the resizing of arguments to functions, so they can be used. A very clear example would be if we wished to add a scalar value to each element in an array of scalars. To do this with vectorization alone would mean we would need another array which replicates the scalar we wish to add for each element in the array we wish to add it to. Broadcasting basically does this for us, as exemplified in Equation 4.

$$\vec{a} + 2 = [a_1 + 2, \ldots, a_n + 2] \tag{4}$$

Array programming languages also often support higher-order operators for use on arrays. An important operator for arrays is fold (or reduce), which applies a binary

function to elements in an array, where one argument accumulates the previous results. It is easy to imagine how such an operator could be used to, for instance, sum all the items in a 1-dimensional array. An important realization is that, since fold only returns the final result, fold can reduce the dimensions of an array by one. In our summation example, we fold a one-dimensional array into a zero-dimensional array, namely a scalar value. Similar to fold is scan, which like fold applies a cumulative binary function to each element in the array, but rather than returning only the result, it returns all intermediate results in an array (with the last element being the final result).

Other important array functions include map, which applies a function to each element in an array. Then, forward permutation (scatter) and backwards permutation (gather), which permute one array into a new one by respectively mapping the indices of the source array to those of the new array or the indices of the new array to those of the source array. Generate, which generates a new array as well, but does by taking the dimensions of the desired array and a function that takes in an index and outputs a value.

We should also not gloss over the actual implementation of these arrays, especially in functional languages where there exists two major ways of constructing arrays [23]. Pull-arrays are the more used of the two, here arrays are represented with a function from an index to a value. In push-arrays, consumers are provided with a method to write into memory. This means that the way that the efficiency of array operations can change based on the array representation. For instance, indexing is faster on pull-arrays while push-arrays are quicker to concatenate. This basically divides the array operations in two camps: push-operations and pull-operations.

These two camps play an important role in a concept of fusion. When we have multiple back-to-back parallel array operations, executing them naively introduces a lot of overhead for reading and writing intermediate values to memory. Instead, fusion allows us to combine these operations together, so we can compute them in one go without the overhead of storing intermediates. However, we cannot just go chaining parallel operations, not all parallel operations fuse together nicely. In fact, pull-array operations only fuse with other pull-array operations, and the same goes for push-array operations. This means for instance that we can fuse multiple scatter operations, but not a scatter and gather operation.

Now the reason for choosing a (functional) array language over a general language is often because we need to process large amounts of numerical data, and arrays are well-suited for parallelism. To be precise, we are talking about data parallelism here. Task parallelism is when two or more computer processes run simultaneously on different processor cores. Data parallelism is when an operation (or a string of operations) is done element-wise on data structure like an array. The parallelism of data parallel processing of these operations on each element, rather than the parallelism of different processes. An important distinction between task and data parallelism, is that while parallel threads in task parallelism can generally start, run, and end independently of each other, data parallelism threads move in lockstep with each other. This is lockstep or synchronous execution means that the execution does not continue until the current operation has been applied to all elements in the array, which may be important if we want to do multiple parallel operations back-to-back. Furthermore, modern GPU architectures are especially well-suited for this type of synchronous parallelism, as graphics processing overlaps in large part with parallel array processing.

A good starting point for the history of functional parallel array programming was

in 1992, with G. Belloch's paper on the parallel array programming language NESL [24]. The language was strongly-typed and had no support for side effects, making it a functional language. The main way to add parallelism was through the inherently data-parallel "vectors" the language introduces in lieu of lists. These vectors could also be nested, and functions could run in nested parallel on these vectors. Another major inclusion was to allow user-defined functions to be run (in parallel) on these vectors, making it possible to write more complex nested data-parallel algorithms than before.

The functional language Haskell, saw the introduction of task-parallelism well before its first official release, through libraries like pH [25]. Some data-parallelism followed [26, 27, 28], but this was limited to applying a function over a flat array. However, in 2001 nested data-parallelism was introduced to Haskell by the NEPAL project by Chakravarty et al. [29]. The paper largely focusses on reimplementing NESL as a Haskell library, but creates a much more expressive data-parallel language doing so. This is because NESL was rather limited in scope, whereas Haskell was already a full-fledged functional programming language. Two important concepts come to the forefront in the NEPAL paper, namely flattening and fusion. Both in NESL and in NEPAL, higher-dimensional nested parallelism is "flattened" to a single distributed parallel operation. In NESL, this meant that data-types had to be limited to tuples and the vectors it introduced, to make sure this flattening operation worked correctly. Since then however, Keller and Chakravarty had shown this flattening transformation could also be applied more generally to cover the full range of types of general programming languages [30, 31, 32]. This allowed them to apply the nested data parallelism of NESL to a more expressive language Haskell with NEPAL. Furthermore, they also showed that in combination with fusion it could produce efficient code for distributed machines [33]. Fusion is where multiple separate parallel operations are combined into a single parallel operation, which greatly improves performance of complicated parallel programs. This is important because many operations on arrays introduce the need for intermediate arrays to be computed. Doing this in parallel leads to more problems, as these implementations rely on gang parallelism, where the parallel threads remain in lockstep with each other [34]. Fusion helps us here, as we can reduce the number of intermediate arrays to be generated, as we can calculate the results of multiple operations at once [35, 36].

All this work culminated in 2007's Data Parallel Haskell (DPH) [37], by Peyton-Jones et al. Its main feature was the parallel array, that like NESL's vectors, was the main way of adding parallelism to a program. However, these parallel arrays could now hold any type, such as other arrays or functions, like Haskell's native (non-parallel) lists. Furthermore, DPH provides parallel variants of Haskell's native list functions, and a parallel alternative to Haskell's list comprehensions. The main difference between Haskell's native lists and DPH's parallel arrays (besides the parallelism) was that evaluating any value in a parallel array would require evaluation on all the array's elements, whereas Haskell as a lazy language would not normally do that. This is to be expected, as parallelism becomes meaningless if it is only applied to a single entry of an array.

Outside of Haskell, a functional array-programming dialect of C was developed: Single Assignment C (SAC) [38, 39, 40]. It would go on to distinguish itself as a functional array programming language in a style more familiar to programmers of imperative languages (like C). The main mechanic in SAC is the with-loop, which takes a generator that dictates a looping mechanism and an operation that dictates the return value. These operations can be functions like "fold" to reduce the rank of an array, or "genarray" to generate new (multidimensional) arrays. Besides the imperative style, the main draw of SAC is that its performance is comparable to Fortran and C, while its programs are generally more concise (for intensive numerical applications.)

In 2010, Keller, Chakravarty, et al. presented a new data-parallelism approach for Haskell in "Regular, Shape-polymorphic, Parallel Arrays in Haskell" [41]. Previous approaches had focussed on irregular arrays, where on array could contain arrays of different lengths. The library Repa, introduced in this paper, was made for regular arrays where arrays of each nested rank are the same size. However, this allows the library to be purely functional and support shape polymorphism. While DPH was purely functional as well, it was not especially performant on regular arrays and it also did not support shape polymorphism. In shape polymorphism, the type of collection is fixed (unlike in type polymorphism), but the shape of the collection is not [42]. For instance, under shape polymorphism a function may be applied to either a flat array, or a 10-dimensional one. While shape polymorphism for functional arrays had been implemented before in SAC, Repa implemented it by embedding it into Haskell's type system, whereas the SAC implementation had required a purpose-built compiler. This also allowed programmers to more easily see and control the shapes of their multidimensional parallel arrays, and build their own shape polymorphic parallel functions.

In 2011, Repa was succeeded by the Accelerate project [43]. Accelerate is a library for Haskell, aimed specifically at bringing parallel array programming to modern GPUs. It mimicked many of Haskell's native list functions with parallel alternatives (that run on the GPU), and used the typed shaped polymorphism from Repa. It also separated "collective" (array) computations and scalar computation by wrapping these in Haskell monads. Here, collective computations could include scalar computations, but not the other way around. This meant excluding nested and irregular data parallelism, which in turn allows Accelerate to efficiently run on GPUs (which are much more constrained than CPUs). It also meant that these arrays could only contain scalars, no functions or other types.

Another interesting example of a parallel array programming language is Remora by Slepak et al. [44] The language, inspired by earlier array programming languages APL [45] and J implements rank-polymorphism. Rank polymorphism is similar to shape polymorphism, but it annotates functions and operators with an array rank they can operate on, and was also present in Repa and Accelerate. Remember that scalars are considered rank 0 arrays, a flat array is rank 1, a matrix is rank 2, et cetera. In rank polymorphism, arguments are transformed (re-ranked) such that they are the rank required for a specific function or operator. Specifically, an operator defined for a certain rank, is automatically defined for any higher rank, because it can be mapped over these higher dimensions. This is subtly different from the more general shape-polymorphism, as rank only refers to the number of dimensions, while shape also contains information on the size of these dimensions. With Remora, Slepak et al. tried to shed some light on the more "murkier corners" of the array-computational model. They do this by generalizing the array-computational model, which then allows them to both address some shortcomings of APL, but also allows them to extend the model to allow arrays of functions and arrays of arguments, which in turn allows for the parallel MIMD (multiple instruction, multiple data) architecture, rather than only SIMD (single instruction, multiple data) parallelism.

In 2017, we got one of the major current functional data-parallel array languages in Futhark [46]. Futhark's design focusses on efficient nested data-parallelism. They do this by using both "aggressive" fusion (fusing as much as possible), followed by flattening (like we saw in NESL). Finally, through some more optimizations, Futhark produces very performant programs. To facilitate this performance however, they do not support higher-order programming, as Futhark only supports up to second-order.

Finally, a more recent parallel array programming language is Dex [47, 48]. Rather than avoiding loops and explicit indexing, like NESL, NEPAL, DPH, and Repa had all done, Dex suggests that these features might introduce more clarity, if only they were implemented correctly. The main idea is to treat index sets as types and arrays as functions. In reality this "index comprehension" can also be seen as functions that return arrays, and allow declaring iteration over multiple dimensions in a single line. Of course, this is the same idea as pull-arrays, a representation also used by Accelerate under the hood. However, the main novelty of Dex is that they use this to make explicit loops, which in turn makes some parallelism opportunities also explicit. Also, when these index comprehensions are presented back-to-back, opportunities for fusion become fairly clear as well. In their paper, they also show that on some benchmark problems, Dex performs similarly to Futhark, as a functional array programming language that was specifically designed to write performant parallel GPU code.

3 Tracing

In the broadest terms, when we trace a program, we track the most basic steps the program takes provided some input. This is relevant for many applications in computer science. For example, certain automatic differentiation (AD) effectively implement the forward-pass as tracing, and then perform the reverse pass on the trace. Tracing is also used in artificial intelligence, where tracing applications can help determine how much memory needs to be allocated, which can speed up training if the model is run multiple times.

However, despite its ambivalence, tracing is rarely properly defined, or defined only for a specific use case. So, in this section we set out to create a more general definition of tracing.

To start, it will help us along to set clear expectations for what we want a tracing function to do. In the simplest terms, we expect a tracing program to take an input program with a set of inputs, and output a "trace". This output trace is defined as a sequence of operations the input program performed on the inputs to get the expected output. A term often used for a trace is a "single-line program": a program without control flow. Clearing control flow like if-then-else statements is only natural: after all, provided some input the program will only walk down one variation of this branching path.

Furthermore, it is also generally accepted that the trace consists of a subset of the types in the input program. Because we are generally more interested in what happens to the data in our program, we can "trace away" functions and data structures. More precisely, say our input program has the types as defined in Equation 5, where we have sum-types as $\tau + \sigma$, product types as $\tau \times \sigma$, functions as $\tau \to \sigma$, literal real numbers, and literal Booleans.

$$\tau, \sigma \coloneqq \tau + \sigma \mid \tau \times \sigma \mid \tau \to \sigma \mid \mathbb{R} \mid \mathbb{B} \tag{5}$$

We can imagine our simplified language, in which we will express our trace – as a language with fewer type formers. By choosing a subset of the type formers in our program, we can indicate which data structures should be traced away. A common option is to keep only "ground types", where we defined a ground type as a type that is

not constructed of other types. Looking at our example in Equation 5, a trace keeping only these ground types would keep only the real numbers and the Booleans as they are not built of other types. Another common option is to keep only continuous types, tracing away all unground and discrete types. Doing that on our type set in Equation 5 would leave us with only the real numbers. This is under the assumption that the discrete types are not actually used as data we are interested in tracing of course, but since tracing will remove all control flow from the program, keeping Booleans and operations on Booleans intact may be meaningless.

The main take-away here is that there is some freedom of choice in what to trace away. What parts we keep and what parts we trace away is very dependent on what information we want to keep in our trace, which in turn is dependent on what our exact goal is for the tracing in the first place.

We can also choose to keep some of our unground types, but then we run into a problem. Say we keep only functions ($\tau \to \sigma$) and real numbers, but our input program contains a function with type $\tau \to (\sigma_1 + \sigma_2)$. This typing is valid in our input program, but no longer valid in our trace, so we find ourselves in a bind. It will be impossible to trace away the sum-type in the output of the function without tracing away the function itself. This is because tracing something away basically means either deconstructing or ignoring it in the trace. For instance, tracing away a tuple, would mean tracing the individual components of that tuple to trace it away. Whereas keeping things in the trace means just keeping them untouched. Therefore, we cannot keep a type like a function $\tau \to (\sigma_1 + \sigma_2)$ in our trace, because we cannot access the sum type without tracing away the function. Of course, we could define a subset $\tau', \sigma' \coloneqq \mathbb{R}$ and then redefine (or add a definition for) our function so that it becomes $\tau' \to \sigma'$ making it safe to trace. This then underlines the rule at work here: we can only keep types that do cannot be constructed of types that are traced away. This is why the ground types are a natural set of types to keep, as they are never constructed from other types.

In a similar vein, we may also encounter operators in our trace that take in or produce types that are not allowed in our trace. For operators that produce a type that is not in our trace, tracing them away is no problem. Since we know we will not be interested in whatever output they produce for our trace, we can simply omit them from the trace altogether. For instance, if we keep only real numbers in our trace like before, an operator returning a Boolean value is of no interest for the trace. However, this is not a simple for operations that take in a type we wish to trace away, yet produce a type we wish to keep in our trace. A simple example of this is the "switch" operator, which takes in a Boolean value and two values of another type, of which it returns one depending on the Boolean value (see Equation 6).

$$\begin{aligned} \texttt{switch}(\top, a, b) &= a \\ \texttt{switch}(\bot, a, b) &= b \end{aligned} \tag{6}$$

While the switch operator looks like it mimics if-then-else statements, it is generally accepted that it does so in a non-lazy way, where both a and b are evaluated before returning either. The main problem here is that we wish to keep operators that produce types we keep in our trace, yet we do not wish (or are not even able to) express the Boolean value in our trace. Now, due to switch statement's likeness to if-then-else, the solution here is pretty clear: only trace the value that gets returned. However, it is not always that easy: as we introduce arrays and array operations in Section 3.4, we will see how operations like mapping on an array need a special solution.

This all is to say that the while we can either ignore or homomorphically copy basic

operations for our trace, sometimes we need a special solution. Mainly because we do not want to lose the information that is needed to execute the trace as a single-line program, even if that means fudging our operations a little. This also means that, while the operations in our trace language might be a subset of the operations in the original expression language, they might contain modified operations

It seems that our tracing definition comes down to a function that takes in a program and an input to that program, and outputs the steps taken by the program run on the input. Where the input program uses some set of types, of which only a subset is kept in the trace, where the types in this subset may not be constructed using types from outside the subset. What now remains is a concrete definition of the output of the tracing program. We have already stated that it should somehow contain the steps done by the input program. The steps we wish to record are generally basic operations like arithmetic operations. But other operations, such as operations on arrays, can also be added depending on the ultimate goal of the tracing. More importantly, as we expect our trace to be akin to a single-line program, we may consider our trace as a series of let-bindings, akin to A-normal form. This means storing each operation as a pair of a unique name or ID and the operation performed (like the name and value of the declarations in a let-binding).

3.1 Tracing Correctness

Before going into specifics on how to implement tracing, it would also be a good idea to formalize when a trace is actually correct. Like we posed before, we start with some program formed from some expression language S, and some input I that is valid for that program. If we wish to resolve a program S on input I, then we would need some evaluation function that produces the expected output O. Now, given some trace language T we can write a tracing function that gives us the trace and output of a specific program and input combination. We can write this out as the two functions `eval` and `trace` in Equation 7.

$$\begin{aligned} \text{eval} &: S \times I \to O \\ \text{trace} &: S \times I \to T \times O \end{aligned} \tag{7}$$

With this we can formalize two criteria for our trace. First, the trace, as a single line program $t \in T$ produced by the trace function needs to produce the correct output. Now, as mentioned before, t might contain transformed operations, that are not present in S. Therefore, we either need to look at traces $t \in S \cap T$, or use a different evaluation function. For now, we will use the former to assert the output criterium in Equation 8. Here we state that for any program s with any input i: if the trace t is also a valid program in S, that the evaluation of t on i should be the same as the evaluation of s on i or the output o we got out of the tracing function.

$$\begin{aligned} &\forall s \in S \\ &\forall i \in I \\ &\text{trace}(s, i) = (t \in T, o \in O) \\ &(t \in S \cap T) \to (\text{eval}(s, i) = \text{eval}(t, i) = o) \end{aligned} \tag{8}$$

Furthermore, tracing a trace t should also return that trace t. This is because we want to find the minimal straight-line program using tracing, and if tracing the trace we found

reduces it somehow to a more minimal program, we know that the original trace was incomplete. This is expressed in Equation 9, where we assert that for some program $s \in S$ and some input $i \in I$, the trace t (produced by tracing s on i), is the same as the trace obtained from tracing t itself.

$$\begin{aligned} &\forall s \in S \\ &\forall i \in I \\ &\text{trace}(s, i) = (t \in T, o \in O) \\ &\text{trace}(t, i) = (t, o) \end{aligned} \tag{9}$$

The above statements, assert that a trace should produce the correct output value as expected from the input program, and that a trace should be its own trace. While these assertions do not say a lot about the nature of the actual trace, they do set some baseline requirements for the trace, and proving the correctness of a trace. This vagueness on the contents of the trace is partly because we cannot really say anything about a trace without dissecting the source program as well, which would bring us to a point very close to actual tracing itself. In another part however, this is because we do not want to make any assumptions what can or cannot be in our trace. While it is likely that some there is significant overlap between S and T, as mentioned, we might need some additions to T to actually be able to trace everything in S correctly. Also, whilst in practice it might be meaningless, a trace where $T = \emptyset$ is in itself not incorrect: any trace would simply be empty. In a similar vein a trace where $S \subseteq T$ would also be meaningless in practice, it is also not wrong: any trace would simply be the same as the source program.

As an additional note, Equation 8 also implies something interesting. If we want our trace to output the same value as the original program, we cannot trace away the type of the original programs output. Say we trace away Boolean values when we are tracing a program that returns a Boolean value, then we find ourselves stuck, because we trace away all operations that produce Boolean values. And of course, if our trace is not allowed to produce any Boolean values, we cannot produce the required output either. Therefore, we must assure that the type of the output is valid in our trace as well.

3.2 Basic Tracing

We now define some basic tracing steps for some arbitrary language. For clarity's sake, we will do this is with Haskell code. To do this we first define a language and values on which we will operate. We do this in Listing 3, where we define a basic lambda calculus. Here the value types are represented as the algebraic data type (ADT) `Value`, where we find constructors for Booleans (`VBool`), real numbers (`VReal`), and functions (`VFunc`). Then we define the four terms of a basic lambda calculus in the `Expression` ADT: application (`EApply`), abstraction (`ELambda`), loose values (`ELift`), and variable reference (`ERef`). To make tracing a little more interesting we also add in if-then-else statements (`EIf`) and binary operators (`EOp2`). For those binary operators, we define four operations in the separate `Op2` ADT: addition (`Add`), equality (`Equ`), multiplication (`Mul`), and inequality (`Neq`). Finally, to make use of variable references, we define an environment as a mapping of strings to values. We interact with this environment in two ways: by inserting values into them, and getting values from them (indexing). The function signatures for these interactions, respectively `insert` and `(!)`, have been included in Listing 3 as well. We can use this language and evaluate it, an example of this has been provided in Appendix A.

```haskell
data Value = VBool Bool | VReal Float | VFunc (Value -> Value)

data Expression
    = EApply  Expression Expression
    | EIf     Expression Expression Expression
    | ELambda String     Expression
    | ELift   Value
    | EOp2    Op2        Expression Expression
    | ERef    String

data Op2 = Add | Equ | Mul | Neq

type Environment = Map String Value

-- Operations on maps:
-- (where Map a b is a mapping from keys of type a to values of type b)
insert :: a -> b -> Map a b -> Map a b
(!) :: Map a b -> a -> b
```

Listing 3: Minimal lambda calculus with added if-then-else and binary operators

With our language in Listing 3, we can almost start tracing. However, we must first decide which parts of the language we keep, and which parts we wish to trace away. In the previous section, we talked about how we can do this by selecting which type formers we wish to keep. In Listing 3, we have practically defined the types of our values by the data constructors present in the `Value` ADT as Booleans, real numbers, and functions. Let us now choose to keep only real numbers in the trace.

We now define a new ADT for traced values in Listing 4. This is only so we can incorporate a name into the values we wish to keep in our trace. These names will help us read the trace, and can be incrementing numbers or something entirely random, as long as they are unique. The basic idea is here to feed the `trace` function a number with which to generate the steps' names from, and increment the number every time we do. However, since this clutters the code while not being very interesting, we will assume we have some function `getName` that provides us with a unique name. Furthermore, it is important to see that we still have Boolean values and functions in our `TValue` ADT, even though we only wish to keep real numbers in our trace. This is because we might still need these values to resolve expressions, even if they never end up in the trace. We might also achieve this by extending our original `Value` ADT (from Listing 3) with traced variants of values, but this is merely a point of preference. Finally, we have also changed the signature of the function value to return a trace as well, as we move on to functions we will see how this works.

First however, with our basic building blocks for tracing set up, lets trace away these boolean values. We do this with the trace function in Listing 5. For now, we will leave out abstraction and application, as it might be easier to talk about tracing away Booleans first.

When we trace away Booleans, like in Listing 5, it is useful to think about where these Boolean values actually come up. In our minimal language from Listing 3, there are only three points: when they are included as literal values, as the input or output to basic operations, or as the conditional in if-then-else statements.

```haskell
data TValue = TBool Bool
            | TReal String Float
            | TFunc (TValue -> (TValue, Trace))

data Traced = TLift TValue | TOp2 Op2 String String

type TEnvironment = Map String TValue

type Trace = [(String, Traced)]

getName :: String
```

Listing 4: Basic trace building blocks

Let us start with the easiest first: literal Boolean values. When we encounter literal values during tracing, and they are of a type we wish to keep for our trace, we simply add their instantiation to the trace (as `TLift` in Listings 4 and 5). This is extremely straightforward: those values might be used by the operations we wish to trace, so they should be included in the trace themselves as well. For values of types we wish to trace away, we simply do not include them in the trace. After all, our trace should be fine without them, as we do not include any operations that require them in our trace, right? For now this seems obvious: if we look at the language in Listing 3, we see that there are no other uses for Booleans values than the use in the equality and inequality operators, and as the conditional in the if-then-else statement. Since we plan to trace these away, we do not appear to need these value instantiations in our trace either. However, at the end of Section 3.1 we already discussed what would happen if our program were to return a Boolean value. And in Section 3.4, we will see how this might not be entirely true when we talk about arrays and operations on arrays like mapping a function.

Tracing (away) simple operations like addition and equality (`EOp2`) are done in a similar vein. If the operation returns a value of a type we wish to keep in our trace, we include the operation in our trace as well. Similarly, if the operation returns a value that we do not wish to keep, we simply do not trace it. Again, if there was an operation that took in a value of a type we do not wish to trace, and returned one that we do wish to trace we run into a problem. Luckily, these operations are not included in our current example.

When we trace an if-then-else statement, we know we have to deal with a Boolean regardless. Luckily for us, we know we only need to trace one of the branches. This means quite simply, that we can ignore the if-then-else statement, and act like the program continued at the branch that is chosen. Since the input is provided, we can resolve the conditional immediately, and then just trace the appropriate branch.

Finally, tracing variable references are simple as well. Currently, the only named variables that occur are those created in lambda abstractions or those that are provided as inputs. But no matter how they are created, variable reference does not require tracing. This is because the trace will reference the values regardless of whether they are instantiated on the spot or somewhere previously. And if they were defined previously, that definition is already in the trace somewhere.

```
trace :: TEnvironment -> Expression -> (TValue, Trace)
trace n (EIf e1 e2 e3) =
    -- Since we e1 should resolve in a Boolean value, we do not need to trace it.
    let v1 = eval n e1
    in  case v1 of
        -- We can check for the type of v1 and its value in one go
        -- We trace only the relevant branch
        (VBool True)  -> trace n e2
        (VBool False) -> trace n e3
        _             -> error ``Type mismatch in trace/EIf''

trace n (ELift v) =
    -- Check if v is a value we would like to trace
    case v of
        -- If yes return the transformed value with its simple trace
        (VReal v) ->
            -- Generate a name for this step and make the TValue
            let s  = getName
                v' = TReal s v
            -- Combine the TValue with a trace of its instantiation
            in  (v', [(s, v')])
        -- If we do not wish to trace something, we can just return the value
        -- with an empty trace.
        (VBool v) -> (TBool v, [])
        -- Instantiation is not allowed for functions, they need to be
        -- abstracted using ELambda
        _         -> error ``Type mismatch in trace/ELift''

trace n (EOp2 op e1 e2) =
    -- We again first trace e1 and e2
    let (v1, t1) = trace n e1
        (v2, t2) = trace n e2
        -- We get a ready name in case we need it
        s = getName
    -- This case syntax allows us to select for the right operator with the
    -- right value types at the same time.
    in  case (op, v1, v2) of
        -- Since add and mul take in reals and produce one too, we trace both
        -- the operation and the origins of v1 and v2
        (Add, TReal s1 a, TReal s2 b) -> (TFloat s (a + b),
            (TOp2 op s1 s2) : t1 ++ t2)
        (Mul, TReal s1 a, TReal s2 b) -> (TFloat s (a * b),
            (TOp2 op s1 s2) : t1 ++ t2)
        -- For operations producing Bools we only return the result, but they
        -- are not traced, and therefor return an empty trace
        (Equ, TBool _  a, TBool _  b) -> (TBool (a == b), [])
        (Equ, TReal _  a, TReal _  b) -> (TBool (a == b), [])
        (Neq, TBool _  a, TBool _  b) -> (TBool (a /= b), [])
        (Neq, TReal _  a, TReal _  b) -> (TBool (a /= b), [])
        _                             -> error ``Type mismatch in trace/EOp2''

-- There is nothing to trace when fetching a variable, but we still need to
-- actually get the value
trace n (ERef s1) = (n ! s1, [])
```

Listing 5: Tracing away Boolean values

3.3 Function Tracing

With our basic tracing established, we can now talk about tracing functions, which are more complicated. It is the tracing of abstracted functions that is the first issue here. The issue is that when we perform an abstraction (as with `ELambda`), there is nothing to trace. In fact, we can see this as an instantiation of a function literal, and when functions are not in our set of types to keep in the trace, this abstraction creates an empty trace. However, leaving it at that would mean we never actually trace the body of the function. Yet, at the time of the abstraction, we also do not yet know the input to the function either, meaning we cannot trace the body at that time. We must instead consider how we delay tracing until the function is actually applied. This is where our notation for `TFunc` (as in Listing 4) comes up. We wish that functions while tracing perform tracing themselves, thus return a `Trace` together with the return value. This is then what we do in the abstraction step: we set the trace on the body of the function as the body of the function we return. Similarly, we also give this tracing function call the environment at the time of abstraction, allowing the function body to access any free variables that were defined at that time. This makes application also very simple: we apply the function, and then just combine the trace of the function's instantiation, with that of the argument, and that of the function's execution. We also trace the function's instantiation, since at the time of application we do not know if the expression that leads to the function does anything else that we might need to trace as well. Finally, this is results in what we see in Listing 6, where we left out any patterns of trace that were already present in Listing 5.

```
1  trace :: TEnvironment -> Expression -> (Value, Trace)
2  trace n (EApply e1 e2) =
3      -- First trace e1 and e2
4      let (v1, t1) = trace n e1
5          (v2, t2) = trace n e2
6      -- Check if v1 actually returns a function
7      in  case v1 of
8          -- Do the application, return the result and the combined trace
9          TFunc f -> let (vf, tf) = f v2
10                    in  (vf, tf ++ t2 ++ t1)
11         _       -> error ``Type mismatch in trace/EApply''
12
13 trace n (ELambda s e1) =
14     -- Define the function, insert value x as variable s into the environment that is currently
15     -- present, and trace the body
16     let f = TFunc (\x -> trace (insert s x) e1)
17     -- Return the function as abstracted function as a value, and no trace
18     in  (f, [])
```

Listing 6: Tracing away functions

```haskell
data Expression
    = ...
    -- The string here is the name of the bound variable
    | ELet String Expression Expression

trace :: Environment -> Expression -> (TValue, Trace)
trace n (ELet s1 e1 e2) =
    -- Evaluate e1 first, then e2 with e1 in its environment
    let (v1, t1) = trace n e1
        (v2, t2) = trace (insert s1 v1 n) e2
    -- Return the value of e2 and the combined trace
    in  (v2, t1 ++ t2)
```

Listing 7: Tracing let bindings

3.3.1 Tracing let bindings

As an additional structure present in functional languages that we might wish to trace, there are let-bindings. Recall that let-bindings are effectively the same a lambda abstractions that are resolved immediately. This makes it extremely easy to resolve them, because we can just trace the let-side of the binding and add it to the environment for the tracing of the right-hand-side.
Adding let-bindings and tracing them is done in Listing 7.

3.4 Array Tracing

Tracing on data structures like arrays provides us with a new problem that revolves around whether we wish to trace arrays away or not. We can see arrays as either structures that contain the data we are really after, which would require us to trace them away, or as data in their own right which we wish to keep in the trace. Both scenarios provide us with interesting challenges.

Let us first talk about tracing arrays away. When we simply view arrays as another computational structure, they are not too complicated to trace away. When initializing an array, we just initialize all the individual values in the array. And when performing operations on items in the array, we instead perform those operations on the individual items again. That is, we do the operation like normal, but denote them as operations on separate items in the trace.

In Listing 8 we first add a arrays and array operations. While we said earlier that constructors in the `TValue` ADT only needed strings for names if they are traced, we need to make an exception for arrays. This is because when working with arrays our expression will never refer to individual values in arrays, only to the array itself (and using its individual values from there). This means that to consistently refer to values that were in arrays in the original expression, we need to give a little more structure to the naming scheme. We do this by taking the name of the array, and adding the index of the item to create a name that is unique yet identifiable. Furthermore, we add in array operations: iota (or range) (`Iota`), generate (`Gen`), indexing (`Idx`), sum (`Sum`), map (`Map`), and folding (reduction) (`Fold`). It should be noted that iota, generate, and indexing take an integer argument as part of their operator. For iota and generate this

is the size of the array to create, and for indexing this is the index to get the value from. While we could allow our language with integers (or by casting floats) to allow using in-language numbers, this really is not all that interesting. If those arguments were part of our trace, it would just mean tracing them like any other item by just ignoring them.

```
1  data Value = ... | VArray [Float]
2
3  data TValue = ... | TArray String [Float]
4
5  data Expression
6      = ...
7      | EOp0 Op0
8      | EOp1 Op1 Expression
9      | EOp2 Op2 Expression Expression
10     | EOp3 Op3 Expression Expression Expression
11
12 data Op0 = Iota Int
13
14 data Op1 = Gen Int | Idx Int | Sum
15
16 data Op2 = ... | Map
17
18 data Op3 = Fold
```

Listing 8: Adding arrays

Now with arrays added to our language, we can actually trace them. This is done in Listings 9 and 14, where we again extend the trace function, leaving out any patterns that remain unchanged.

First off, when encountering a literal array, or creating one with the iota operator, we need to initialize every individual value. This is fairly simple, it just requires us to walk through the array and initialize every value like when we were initializing literal real values.

Indexing, in this mode, is equal to variable reference due to our naming scheme. This means then that we do not need to trace anything here.

The sum operator is a little more in-depth, as shown in Listing 11. However, this is a lot of code for a very simple principle, and a couple edge cases. The principle is, add the first two values in the array together, and then every following item to that result and so on. And we have edge cases for singleton and empty arrays. It is worth explicitly stating that every addition done whilst summing the array gets its own unique name and step in the trace. This means that an operation that is single step the original program, explodes to a bunch of steps (the length of the array minus one) in the trace. This is because we decided to trace away arrays, and we will see later on how we save ourselves from this by not tracing away arrays.

The map operator is funky in a way similar to sum. In essence, we take each item in the array and apply it to the function as expected. However, we run into a little problem with our naming scheme. For map, the items, once mapped on, are placed back into a new array. This means that, according to the scheme we laid out, the items in this array should be named in reference to the new array, however this is not something the call to trace in the function body considers. Luckily we can resolve this by renaming the

```haskell
trace :: TEnvironment -> Expression -> (TValue, Trace)
trace n (ELift v) =
    case v of
        -- Tracing for reals and Booleans remain unchanged
        (VReal  v) -> ...
        (VBool  v) -> ...
        (VArray v) -> let s = getName
                      -- For traceArrayLift see Listing 12
                      in  (TArray s v, traceArrayLift s v 0)

-- With only iota, we could write this a little more curtly, but for clarity we leave it like this
trace n (EOp0 op) =
    case op of
        (Iota r) ->
            let s = getName
                -- Define an array of size r, then lift is using traceArrayLift again
                v = [0.0 .. (r - 1)]
                -- For traceArrayLift see Listing 12
            in  (TArray s v, traceArrayLift s v 0)

trace n (EOp1 op e1) =
    -- Again we first trace e1, and we get a name ready as well
    let (v1, t1) = trace n e1
        s = getName
    in  case (op, v1) of
        (Gen r, TFunc f) ->
            let v  = [0.0 .. (r - 1)]
                tg = traceArrayLift s v 0
                (vm, tm) = traceArrayMap f s v 0
            in  (vm, tm ++ tg)
        -- Indexing is like variable reference, we do not need to add to the trace,
        -- but we need to create the name to be consistent
        (Idx i, TArray s1 v) ->
            -- Get the actual item using indexing (!!)
            let x  = v !! i
                s' = s1 ++ '!' : show i
            in  (TReal s' x, t1)
        -- For traceArraySum see Listing 11
        (Sum, TArray s1 v) ->
            let (vs, ts) = traceArraySum s1 v 0
            -- We must not forget to add the trace of e1 to our trace here
            in  (vs, ts ++ t1)

-- See Listing 14 for trace on EOp2 and EOp3.
```

Listing 9: Tracing away arrays

```
trace n (EOp2 op e1 e2) =
    -- Again we first trace e1 and e2, and we get a name ready as well
    let (v1, t1) = trace n e1
        (v2, t2) = trace n e2
        s        = getName
    in  case (op, v1, v2) of
        ...
        (Map, TFunc f, TArray sa va) ->
            -- For traceArrayMap see Listing 12
            let (vm, tm) = traceArrayMap f sa va 0
            -- Combine the traces
            in  (vm, tm ++ t1 ++ t2)

trace n (EOp3 op e1 e2 e3) =
    -- Again we first trace e1, e2, and e3, and we get a new name ready
    let (v1, t1) = trace n e1
        (v2, t2) = trace n e2
        (v3, t3) = trace n e3
        s        = getName
    in  case (op, v1, v2, v3) of
        (Fold, TFunc f, TReal {}, TArray {}) ->
            -- For foldFunction and traceArrayFold see Listing 13
            let (vf, tf) = traceArrayFold (foldFunction f) v2 v3 0
            -- Combine the traces
            in  (vf, tf ++ t1 ++ t2 ++ t3)
```

Listing 10: Tracing away arrays in maps and folds

returned value from that function, and changing the name in the trace. The signature of a function that does this is also included at the end of Listing 12, but its exact implementation is not of importance here.

Finally, the generate operation can be expressed as an iota operation followed by a map operation. The iota operation provides us with the indices of the array to generate, and we can re-use the code for map for mapping the generator function over these indices. This is also how we implemented it in Listing 9, using `traceArrayLift` from iota and `traceArrayMap` from map.

While the concepts behind tracing away arrays are hopefully not too difficult to understand, it should be obvious from Listings 12 and 11 that the implementation becomes more complex. Now while that is not really a problem, we should really note that the trace becomes messier as well. This is especially problematic if we actually want to read the trace to see what is going on: not impossible, but also not pleasant, especially with large arrays. So perhaps we are tempted to keep arrays in the trace instead, or perhaps we are interested in the trace of arrays specifically.

Luckily for us, in large parts tracing while keeping arrays is fairly easy. This is because we can treat most operations like how we treated operations for real numbers. This has been done in Listing 14, except for generate, map, and fold, where we replace the tracing patterns from Listing 9.

In our current language, the main point of difficulty and interest is the generate, map, and fold operations. They take in a function, which is not a type we wish to keep in our trace, however they produce an array which we wish to keep in our trace. While we

```haskell
-- traceArraySum starts the trace, and traceArraySum' completes it
-- This is necessary because we do not know the number of items in the array
-- traceArraySum takes only the array to sum
traceArraySum (TArray _ []) =
    let s = getName
        v = TReal s 0
    -- The sum of an empty array means just lifting the value 0
    in  (v, [(s, TLift v)])

traceArraySum (TArray _ [x]) =
    let s = getName
        v = TReal s x
    -- The sum of a singleton array is just that one value
    in  (v, [(s, TLift v)])

traceArraySum (TArray sa (x:y:z)) =
    -- When summing on a larger array, the first sum is of the first two items
    let sx = sa ++ ''!0''
        sy = sa ++ ''!1''
        s  = getName
        v  = TReal s (x + y)
        -- Get the result, and the trace of the rest of the array with traceArraySum'
        (rv, rt) = traceArraySum' (TArray sa z) 2 v
    -- Return the final result, but do not forget the trace of the first sum
    in  (rv, (s, TOp2 Add sx sy) : rt)

-- traceArraySum' takes the array we sum over, the current index, and the last calculated value
traceArraySum' :: TValue -> Int -> TValue -> (TValue, Trace)
-- When we are done, return the value
traceArraySum' (TArray _ []) _ v = (v, [])

traceArraySum' (TArray sa (x:xs)) i (TReal sr r) =
    -- Get the name for this item
    let sx = sa ++ '!' : show i
        -- Get the name for this addition step
        s  = getName
        -- Get the result of the rest of the array
        (v, t) = traceArraySum' (TArray sa xs) i (TReal s (x + r))
    -- Return the final result, and add this steps addition to the trace
    in  (v, (s, TOp2 Add sx sr) : t)
```

Listing 11: Tracing the sum operator

```haskell
-- traceArrayLift takes the name of the array, the contents, and the current index
traceArrayLift :: String -> [Float] -> Int -> Trace
-- Empty lists get no trace
traceArrayLift _ []     _ = []
traceArrayLift s (x:xs) i =
    -- Create the name for this item from the array's name and the current index
    let s' = s ++ '!' : show i
    -- Trace x as a single real number
        tx = TLift (TReal s' x)
    -- Trace the rest of the array
        txs = traceArrayLift s xs (i + 1)
    -- Return the combined trace
    in  tx : txs

-- traceArrayMap takes the function to map, the name of the old array, the name of the new array,
-- the contents of the old array, and the current index
traceArrayMap :: (TValue -> (TValue, Trace)) -> String -> String -> [Float]
    -> Int -> (TValue, Trace)
traceArrayMap _ _  sn []     _ = (TArray sn [], [])

traceArrayMap f so sn (x:xs) i =
    -- Get the current value from the array with the right name
    let current  = TReal (so ++ '!' : show i) x
    -- Get the result from the function application
        (fv, ft) = f current
    -- Get the results from the rest of the array
        (xsv, xst) = traceArrayMap f so sn xs (i + 1)
    -- To add to the TArray and to rename fv we use this case-of statement
    in  case (fv, xsv) of
        (TReal s' v, TArray _ xsv') ->
            -- Add this item to the new array
            let vn = TArray sn (v : xsv')
            -- Rename fv in the function trace to the correct name
                ft' = rename s' (sn ++ '!' : show i) ft
            -- Finally return the new array and the combined trace
            in  (vn, ft' ++ xst)

rename :: String -> String -> Trace -> Trace
```

Listing 12: Tracing array instantiation and array mapping

```haskell
-- foldFunction makes a binary function out of two nested lambda functions
foldFunction :: (TValue -> (TValue, Trace))
    -> (TValue -> TValue -> (TValue, Trace))
foldFunction f x y = case f x of
    (TFunc g, tf) -> let (vg, tg) = g y in (vg, tf ++ tg)
    _             -> error ``Type mismatch in foldFunction''

-- Traces all steps in a simple left-to-right fold
traceArrayFold :: (TValue -> TValue -> (TValue, Trace)) -> TValue
    -> TValue -> Int -> (TValue, Trace)
-- When the array is empty, return just the identity value and an empty trace
traceArrayFold f z (TArray _  [])       _ = (z, [])

traceArrayFold f z (TArray sa (vx:vxs)) i =
        -- Create a current value
    let x          = TReal (sa ++ `!' : show i) vx
        -- Do the folding step
        (vf,  tf)  = f z x
        -- Continue for the rest of the array
        (vxs, txs) = traceArrayFold f vf (TArray sa vxs) (i + 1)
        -- Combine the trace for the rest of the array with this step's trace
    in  (vxs, tf ++ txs)
```

Listing 13: Tracing array folding

might be tempted to just discard the function component, we cannot do that because it provides the trace from the original array to the new array. Without that information our trace is no longer a functional (straight-line) program.

The intuitive way to solve this, the naïve method, would be to attach an array of traces to the map operator, so they can be followed to derive the correct results. Similarly, we can do this for the generate and fold operations. To easily do this we extend our `Traced` ADT with a special map constructor (`TMap`), and a special fold constructor (`TFold`). We show this in Listings 15 and 16. We will also express our generate operation as a combination of the iota and map operations here, which saves us from writing a special case for generate. The traces in the `TMap` constructor correspond with the application of the function to be mapped to the individual item, for each item. The string references the array the map is performed on. Meanwhile, the `TFold` operator references the folding process as a single sub-trace, and also the name of the map it is executed on.

Now, while the naïve way for maps is fine in functionality, it does again create some overhead (a trace for each item in the array) by splitting the trace into multiple smaller traces. And if the function is the same for every item in the array, we may find ourselves saving a lot of redundant data. Now, this may be necessary: at the time we map a function over an array, we do not know if it will act the same for every input. Perhaps there is some control flow in the function body that checks if a number is even, or a factor of three, or something else entirely. In such a case, having a trace for each item may be strictly necessary. However, it also highlights for which functions it may not be: functions without control flow or branching. After all, these functions are little straight-line programs, and should act the same no matter on what input they are applied (except for producing a different result, of course). Writing a function that checks if the body of a lambda abstraction contains branching is very simple for this language: currently the only expression term that can introduce branching is the if-then-else statement.

```
trace :: TEnvironment -> Expression -> (TValue, Trace)
trace n (ELift v) =
    case v of
        (VReal  v) -> ...
        (VBool  v) -> ...
        (VArray v) ->
            let s = getName
            -- Literal lifting of arrays becomes real simple
            in  (TArray s v, [(s, TLift (TArray s v))])

trace n (EOp0 op) =
    case op of
        (Iota r) ->
            let s = getName
                v = [0.0 .. (r - 1)]
                -- Iota again becomes very similar to literal array lifting
            in  (TArray s v, [s, TLift (TArray s v)])

trace n (EOp1 op e1) =
    -- We trace e1 first, and create a name just in case
    let (v1, t1) = trace n e1
        s = getName
    in  case (op, v1) of
        (Idx i, TArray s1 v) =
            let x  = v !! i
                s' = s1 ++ '!' : show i
            -- Now we trace arrays, indexing becomes more relevant to add to our trace,
            -- as the individual item has not been defined before
            in  (TReal s' x, (s', TOp1 op s1) : t1)
        -- Sum becomes very simple, just apply it to the array
        (Sum, TArray s1 v) = (TReal s (sum v), (s, TOp1 Sum s1) : t1)
```

Listing 14: Tracing whilst keeping arrays

```haskell
data Traced
    = ...
    | TMap  [Trace] String
    | TFold Trace   String String

trace :: TEnvironment -> Expression -> (TValue, Trace)
trace n (EOp1 op e1)    =
    -- We first trace e1, and generate a name
    let (v1, t1) = trace n e1
        s = getName
    in  case (op, v1) of
        ...
        (Gen r, TFunc f) ->
            let tg = (s, TLift (TArray s [0 .. (r - 1)]))
                s' = getName
                (vs, ts) = traceMapNaive f s s' [0 .. (r - 1)] 0
            -- The trace becomes the iota followed by the map, explained below
            in  (vs, (s, TMap ts sa) : tg : t1)

trace n (EOp2 op e1 e2) =
    let (v1, t1) = trace n e1
        (v2, t2) = trace n e2
        s = getName
    in  case (op, v1, v2) of
        ...
        (Map, TFunc f, TArray sa va) ->
            let (vs, ts) = traceMapNaive f sa s va 0
            -- The trace becomes TMap, the collection of traces ts, on the old array v2 (with
            -- name sa)
            in  (vs, [(s, TMap ts sa)])

-- traceMapNaive takes in the function to be mapped, the name of the old array, the name of the
-- new array, the contents of the old array, and the current index
traceMapNaive :: (TValue -> (TValue, Trace)) -> String -> String -> [Float]
    -> Int -> (TValue, [Trace])
-- A map over an empty array returns the empty array and no traces
traceMapNaive _ _  sn []     _ = (TArray sn [], [])

traceMapNaive f so sn (x:xs) i =
    -- Create specific names for the old and new value
    let old = so ++ '!' : show i
        new = sn ++ '!' : show i
        -- Apply the function, getting the value for x and its trace
        (xv, xt) = f (TReal old x)
        -- Apply the function for the rest of the map
        (xsv, xst) = traceMapNaive f so sn xs (i + 1)
        -- We use a case-of statement to append xv to xsv and to rename xv in xt
    in  case (xv, xsv) of
        (TReal s' v, TArray _ vs) ->
            let xt' = rename s' new xt
            in  (TArray sn (v : vs), xt' : xst)
```

Listing 15: Tracing generate and map while keeping arrays, naïvely

```
trace :: TEnvironment -> Expression -> (TValue, Trace)
...
trace n (EOp3 op e1 e2 e3) =
    -- Trace the expression to operate on
    let (v1, t1) = trace n e1
        (v2, t2) = trace n e2
        (v3, t3) = trace n e3
    in  case (v1, v2, v3) of
        (TFunc f, TReal s2 _, TArray s3 _) ->
            -- We use the earlier fold tracing function
            let (vf, tf) = traceArrayFold (foldFunction f) v2 v3
            in  (vf, TFold tf s2 s3 : t1 ++ t2 ++ t3)
        _                                   ->
            error "Type mismatch in trace/EOp3"
```

Listing 16: Naive fold tracing

Unfortunately, we cannot check that at the moment when we trace a map operator. This is because any function here would already have been abstracted to a `TFunc` value. So, we would need to check for branching when we are abstracting the function, and we also need a way to convey if a specific instance of `TFunc` contains branching or not. We write branch-checking into functions in Listing 17. For most terms we can just commute the branch checking to the arguments of that term, but there are a couple exceptions. If-then-else statements are the definition of branching in our language, so they return 'true', and no branching can occur in literal instantiation (`ELift`) or nullary operators (`Op0`) (literal instantiation can also be rewritten as a nullary operator), so they always return 'false'. Only for variable reference, which may return a value without actually providing a code to check, we need to see if the value is a function, and whether it has the branching flag set or not. This works because we set the branching flag when functions are defined using abstraction, and because functions may not be entered as literals.

With our branch checking defined we still need to talk about how we actually apply that and make a trace for map that requires less information. The basic idea here is that we can essentially perform vectorization of our function on the array in our trace: we rewrite the trace such that the function is "applied" to the whole array, rather than its individual items. Now without support for this in our language, this basically amounts to syntactic sugar in our trace, however it will provide us with a much clearer trace. This has been done in Listing 18, where we again add a map operator to our `Traced` ADT. This is because we may need to use the naïve method if a function contains branching, and we cannot vectorize it. In Listing 18 we still use `traceMapNaive` to actually map over our array. This is because we need to get the value of the array regardless, and our function value (`TFunc`) will return traces regardless if we need them or not. Then we can just take the first trace returned by the naïve map tracing, and rename all references to the first item of both the new and old arrays, to references of the whole old and new arrays respectively. For this end we define a function `deepRename` at the end of Listing 18. Like with the renaming function in Listing 12, the implementation of this function is not all that interesting: since all a `Trace` object is, is a list of tuples with a name that may need renaming and a `Traced` constructor referencing zero to two strings that may need renaming. All `deepRename` would do is go over these items and rename any occurrences it finds.

```haskell
data TValue
    = ...
    -- Add a branching flag to TFunc
    | TFunc Bool (TValue -> (TValue, Trace))

branchCheck :: TEnvironment -> Expression -> Bool
-- Encountering an if-else-statement means a encountering a branch
branchCheck _ (EIf _ _ _) = True

branchCheck n (EApply e1 e2) = branchCheck n e1 || branchCheck n e2
branchCheck n (ELambda _ e1) = branchCheck n e1
branchCheck n (ELet _ e1 e2) = branchCheck n e1 || branchCheck n e2
-- ELift is always false, because lifting functions is not allowed
branchCheck _ (ELift) = False
branchCheck _ (EOp0 _) = False
branchCheck n (EOp1 _ e1) = branchCheck n e1
branchCheck n (EOp2 _ e1 e2) = branchCheck n e1 || branchCheck n e2
branchCheck n (EOp3 _ e1 e2 e3) =
    branchCheck n e1 || branchCheck n e2 || branchCheck n e3

-- If our variable contains a function we need to check what it has the branching flag set to
branchCheck n (ERef s1) = case n ! s1 of
    (TFunc b _) -> b
    _           -> False
```

Listing 17: Checking for branches

We can also be more efficient with fold, rather than doing a sequential left-to-right fold. Like vectorized maps, we can execute folds using data parallelism if the folding function is vectorizable. A data parallel fold is a little more in-depth than a vectorized mapping operation. Mainly because, at some point, fold requires some sequential steps. However, it is not too complicated: we segment the original array over a number of threads, and let those run sequentially. Then we gather those results in another array, and sequentially fold over that array to finish the fold. Of course, for this to work properly every time, the function used in the fold needs to be associative, so it does not matter we run the fold out of order. We implement this two-step data-parallel fold in Listings 19 and 20, where we also introduce a new constructor to the `Traced` data type, namely `FoldV`. We see in Listing 20 how we distribute the work over the number of threads available to us (represented by the `threads` variable). For each thread we still use the original function (that traces away arrays), because it performs the (partial) fold, and produces the correct trace. Then, when we have no threads left, we combine our results and trace the fold over that as well. It should be clear that the code in Listings 19 and 20 do not run the code using parallelism. Instead, they are just a sequential implementation meant to show off how we could build a parallel variant.

What is important to take away from the shenanigans with the generate, map, and fold operators is that, whilst our definitions and correctness assertions from Sections 3 and 3.1 gave us some guidance, there is ultimately no single way to trace everything. The most important factor here is to keep reminding ourselves of the information we wish to keep in the trace. Not only the value types, but we also need the information needed to actually run the trace as a program. Keeping this in mind, it becomes much more obvious how to trace these operations.

```
data Traced
    = ...
    -- We leave the naive TMap untouched
    | TMap  [Trace] String
    -- And add a new one for vectorized traces
    | TMapV Trace   String

trace :: TEnvironment -> Expression -> (TValue, Trace)
trace n (ELambda s1 e1) =
    -- We add branch checking when we handle abstraction
    let b = branchCheck e1
        f = TFunc b (\x -> trace (insert s x) e1)
    in  (f, [])

trace n (EOp1 op e1) =
    let (v1, t1) = trace n e1
        s = getName
    in  case (op, v1) of
        ...
        (Gen r, TFunc b f) ->
            let tg = (s, TOp0 (Iota r))
                s' = getName
                (vs, ts) = traceMapNaive f s s' [0 .. (r - 1)] 0
            in  if   b
                then (vs, (s', TMap ts s) : tg : t1)
                else let t' = vectorizeTrace s' s (head ts)
                     in  (vs, (s, TMapV t' s) : tg : t1)

trace n (EOp2 op e1 e2) =
    let (v1, t1) = trace n e1
        (v2, t2) = trace n e2
        s = getName
    in  case (op, v1, v2) of
        ...
        (Map, TFunc b f, TArray sa va) ->
            -- We first get the result array (and all the traces) using the naive method
            let (vs, ts) = traceMapNaive f sa s va 0
            in  if   b
                -- If the function contains branching, use the naive method
                then (vs, (s, TMap ts sa) : t1 ++ t2)
                -- Otherwise use the new method
                else let t = vectorizeTrace sa s (head ts)
                     in  (vs, (s, TMapV t sa) : t1 ++ t2)

vectorizeTrace :: String -> String -> Trace -> Trace
-- Rename the references to individual items to the whole array
vectorizeTrace so sn t = deepRename iso so (deepRename isn sn t)
    -- The names for the individual items in this trace
    where iso = so ++ ``!0''
          isn = sn ++ ``!0''

deepRename :: String -> String -> Trace -> Trace
```

Listing 18: Array mapping with trace vectorization

```haskell
data Traced
    = ...
    -- We leave the original TFold untouched
    | TFold Trace String String
    -- And add a new one for vectorized folds
    | TFold Trace String Trace  String String
    -- And we add a new helper constructor used for foldv
    | TJoin [String]

trace :: TEnvironment -> Expression -> (TValue, Trace)
...
trace n (EOp3 op e1 e2 e3) =
    let (v1, t1) = trace n e1
        (v2, t2) = trace n e2
        (v3, t3) = trace n e3
    in  case (v1, v2, v3) of
        (TFunc b f, TReal s2 v2', TArray s3 _) ->
            if   b
                 -- If branching is present, use the old method
            then let (vf, tf) = traceArrayFold f v2 v3
                 in  (vf, TFold tf s2 s3 : t1 ++ t2 ++ t3)
                 -- If not, we will use the segmented fold,
                 -- which returns the final value, the traces for the first step,
                 -- the traces for the second step, and the name of the join variable
            else let (vf, tf1, js, tf2) =
                           segmentedFold (foldFunction f) v3 s2 v2'
                 in (vf, TFoldV tf1 js tf2 s2 s3 : t1 ++ t2 ++ t3)

segmentedFold :: (TValue -> TValue -> (TValue, Trace))
    -> TValue -> String -> [Float]
    -> (TValue, [Trace], String, Trace)
-- See Listing 20
```

Listing 19: Data parallel fold tracing

```
segmentedFold :: (TValue -> TValue -> (TValue, Trace))
    -> TValue -> String -> [Float]
    -> (TValue, [Trace], String, Trace)
segmentedFold f z sa xs =
    controller threads [] xs [] []
    where
        controller :: Int -> [Trace] -> [Float] -> [Float]
            -> [String] -> (TValue, [Trace], String, Trace)
        -- When all threads are done, combine them in to one
        controller 0 ts _   rs srs =
            let s  = getName
                z' = TReal (s ++ ``!0'') (head rs)
                a' = TArray s (tail rs)
                (vc, tc) = traceArrayFold f z' a' 1
                js = getName
                tc' = (js, TJoin srs) : tc
            in  (vc, ts, js, tc')

        controller t ts xs' rs srs =
            -- If we can evenly distribute the rest of the array over the next threads
            | mod (length xs' + 1) t == 0 =
                if   length xs' == length xs
                then let size = length xs' `div` threads
                         sarr = take (size - 1) xs'
                         (vf, tf) = traceArrayFold f z (TArray sa sarr) 0
                     in  case vf of
                         TReal sr r -> controller (t - 1) (tf : ts)
                            (drop (size - 1) xs') (r : rs) (sr: srs)
                else let size = length xs' `div` threads
                         sarr = take size xs'
                         l    = length xs - length xs'
                         z'   = TReal (sa ++ `!' : show l) (head sarr)
                         (vf, tf) = traceArrayFold f z' (TArray sa (tail sarr)) l
                     in  case vf of
                         TReal sr r -> controller (t - 1) (tf : ts)
                            (drop size xs') (r : rs) (sr: srs)
            | otherwise              =
                if   length xs' == length xs
                then let size = length xs' `div` threads
                         sarr = take size xs'
                         (vf, tf) = traceArrayFold f z (TArray sa sarr) 0
                     in  case vf of
                         TReal sr r -> controller (t - 1) (tf : ts)
                            (drop size xs') (r : rs) (sr: srs)
                else let size = length xs' `div` threads + 1
                         sarr = take size xs'
                         l    = length xs - length xs'
                         z'   = TReal (sa ++ `!' : show l) (head sarr)
                         (vf, tf) = traceArrayFold f z' (TArray sa (tail sarr)) l
                     in  case vf of
                         TReal sr r -> controller (t - 1) (tf : ts)
                            (drop size xs') (r : rs) (sr: srs)
```

Listing 20: Segmented fold

4 Automatic Differentiation

Tracing is useful in many applications, one of which is Automatic Differentiation (AD). Recall how in AD we wish to calculate the derivative of a computer program. To do this (in reverse-mode) we wish to calculate the adjoints for the inputs. However, to calculate these adjoints, we would first need to calculate the adjoints for the individual computational steps in the program that contribute to an input's sensitivity. Of course, it is these steps that are represented in the trace of a program. In fact, there is a really close relation between the tapes discussed in Section 2.1 and tracing.

The main difference between the tape used for AD and a regular trace as laid out in Section 3, is the lack of intermediate values in the latter. However, provided a trace, we could simply calculate these intermediate values. Even better is just storing the intermediate values while we trace a program; this is not really any extra work because these intermediate values are calculated by the tracing function already. Consider our trace definition in Listing 4 as a list of tuples consisting of strings as identifiers and a data constructor denoting the action taken. We could just add intermediate values to this structure, but we will soon find this not to be quite enough.

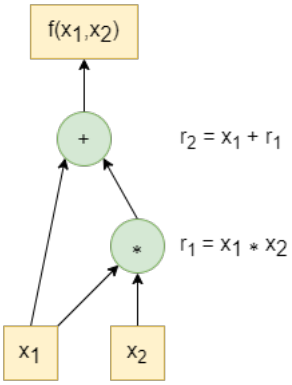

Fig. 1: Computational graph of $f(x_1, x_2) \coloneqq x_1 + (x_1 \times x_2)$

For instance, look at the computational graph in Figure 1 for $f(x_1, x_2) \coloneqq x_1 + (x_1 \times x_2)$. Now, let us say $x_1 = 5$, and $x_2 = 3$, and trace it using the method from Section 3. This gives us the trace as `trace_result` in Listing 21. This trace is very straightforward: x_1 and x_2 are assigned their values, and the multiplication is used in the addition, so it shows up first. Now, let us look at the partial derivatives of f in Equation 10, as we would calculate them using chain rule. In Equation 11 we see which calculations we need to perform, we define the partial derivatives or "adjoints" of a variable r_i as $\bar{r}_i$.

```
f :: Value -> Value -> Expression
f x1 x2 = ELet "x1" (ELift x1) (
    ELet "x2" (ELift x2) (
        EOp2 Add (ERef "x1") (
            EOp2 Mul (ERef "x1") (ERef "x2")
        )
    ))

trace_result :: (TValue, Trace)
trace_result = (TReal "r2" 20.0, [
    ("x1", TLift (TReal "x1" 5.0)),
    ("x2", TLift (TReal "x2" 3.0)),
    ("r1", TOp2 Mul "x1" "x2"),
    ("r2", TOp2 Add "x1" "r1")
])
```

Listing 21: DSL definition of f and its trace

We also assume here that the "seed" value (the value of $\bar{f}$) is one.

$$\begin{aligned}
\frac{df}{d\vec{x}} = \nabla f &= \begin{bmatrix} \frac{\partial r_2(x_1,r_1)}{\partial x_1} \\ \frac{\partial r_2(x_1,r_1)}{\partial x_2} \end{bmatrix}^\mathsf{T} \\
&= \begin{bmatrix} \frac{\partial x_1}{\partial x_1} + \frac{\partial r_2(x_1,r_1)}{\partial r_1} \cdot \frac{\partial r_1(x_1,x_2)}{\partial x_1} \\ \frac{\partial x_1}{\partial x_2} + \frac{\partial r_2(x_1,r_1)}{\partial r_1} \cdot \frac{\partial r_1(x_1,x_2)}{\partial x_2} \end{bmatrix}^\mathsf{T} \\
&= \begin{bmatrix} 1 + \frac{\partial r_2(x_1,r_1)}{\partial r_1} \cdot \frac{\partial r_1(x_1,x_2)}{\partial x_1} \\ 0 + \frac{\partial r_2(x_1,r_1)}{\partial r_1} \cdot \frac{\partial r_1(x_1,x_2)}{\partial x_2} \end{bmatrix}^\mathsf{T} \\
&= \begin{bmatrix} 1 + 1 \cdot \frac{\partial r_1(x_1,x_2)}{\partial x_1} \\ 0 + 1 \cdot \frac{\partial r_1(x_1,x_2)}{\partial x_2} \end{bmatrix}^\mathsf{T} \\
&= \begin{bmatrix} 1 + x_2 \\ x_1 \end{bmatrix}^\mathsf{T}
\end{aligned} \tag{10}$$

$$\begin{aligned}
\bar{f} = \bar{r}_2 &= 1 \\
\bar{r}_1 &= \bar{r}_2 \times 1 \\
\bar{x}_2 &= \bar{r}_1 \times x_1 \\
\bar{x}_1 &= \bar{r}_2 \times 1 \\
&\quad + \bar{r}_1 \times x_2
\end{aligned} \tag{11}$$

With our trace and derivative operations defined, we can now look at how we would get from one to the other. It is important to start at the output of the program, and since the trace function we defined in Section 3 provides us with the named output, we know where to start on our reverse pass. In this case, that would be r_2. As the final value in the primal calculation is the output of the program, its adjoint will be equal to the adjoint of the program or the seed value. This is why Equation 11 posits $\bar{f} = \bar{r}_2$.

Since we are currently working in reverse execution order, we can just use $\bar{r}_2$ to calculate $\bar{x}_1$ and $\bar{r}_1$ directly. It should be reiterated that the trace does not encode any explicit information on the order of operations taken while tracing. It is of course a list that was built up one operation at the time, but relying on this forces us to do our reverse pass

linearly through the trace, which would prevent some task parallelism opportunities. Furthermore, while we can also deduce some order from the naming of the intermediate steps (e.g. r_1 was done before r_2), we should not do this programmatically, because we wish to reserve parallelism opportunities, but also because some intermediate steps might be hidden in the sub-trace of a map. Luckily, we can also discover the "ancestors" of any step in the trace by looking at the traced operation. For r_2 the traced operation was `TOp2 Add "x1" "r1"`, so we know that for our reverse pass, we next want to look at x_1 and r_1, as their adjoints (or part of them) rely on the value of $\bar{r}_2$ (which we can also see in Equation 11). For now, we will gloss over how we decide which ancestor adjoint to compute first, and just look at the adjoint of r_1.

We know that $\bar{r}_1$ is dependent on $\bar{r}_2$, but how exactly is defined by the operation that produced r_2, which in this case is addition. Now, addition is really simple, as the derivative of addition of two values is the addition of the derivatives of those values. See Equation 12, where we calculate the adjoint $\bar{r}_1$ and see how this addition just resolves to 1.

$$\begin{aligned}
\bar{r}_1 &= \bar{r}_2 \cdot \frac{\partial r_2(x_1, r_1)}{\partial r_1} \\
&= \bar{r}_2 \cdot \frac{\partial (x_1 + r_1)}{\partial r_1} \\
&= \bar{r}_2 \cdot \left(\frac{\partial x_1}{\partial r_1} + \frac{\partial r_1(x_1, x_2)}{\partial r_1} \right) \\
&= \bar{r}_2 \cdot (0 + 1) \\
&= \bar{r}_2
\end{aligned} \tag{12}$$

We can again find the ancestors of r_1 by looking at the trace, where we find x_1 and x_2. Let us look at x_2 first. $\bar{x}_2$ is dependent on $\bar{r}_1$, which we just calculated, but rather than an addition (like r_2), r_1 is a multiplication. We mentioned in Section 2.1, in Equation 1, how the derivative of a multiplication uses both the primal part and the derivative part of a number. To get $\bar{x}_2$ we realize (as is visible in Equation 11 as well), that we need the primal value of x_1. We mentioned before we needed the intermediate values, and this is why. Multiplication is not the only operation that requires a primal component, but it is a prime example. We see in Equation 13 how this adjoint resolves to use the primal component x_1.

$$\begin{aligned}
\bar{x}_2 &= \bar{r}_1 \cdot \frac{\partial r_1(x_1, x_2)}{\partial x_2} \\
&= \bar{r}_1 \cdot \frac{\partial (x_1 \cdot x_2)}{\partial x_2} \\
&= \bar{r}_1 \cdot x_1
\end{aligned} \tag{13}$$

Now would also be a good time to quickly reflect on the difference between the tangent (from forward-mode AD) and the adjoint. In forward-mode AD, the operation taken to produce some variable, would influence the tangent of that variable. This is somewhat intuitive, r_1 is a multiplication, and its tangent is $\dot{r_1} = \dot{x_1} \times x_2 + \dot{x_2} \times x_1$. However, this is not the case for reverse-mode AD. In reverse-mode, we see that this information gets passed on to the adjoints of the variable used by the operation, rather than the variable it produced. It should be clear why: the tangents denote how the variable is influenced by a change in the inputs, while an adjoint denotes how its corresponding variable influences the outputs. It is important to closely observe this, mainly for implementation purposes: we want to calculate (part of) the adjoint before we actually arrive at that step in the trace. To calculate $\bar{x}_2$ we need to know what variable x_2 was multiplied with (namely

x_1). This means that if we do not want to search through our trace looking for references (to x_2 for example) every time, it would be better to calculate (the relevant part of) $\bar{x}_2$ while we still see how it is being used.
This then also bring us neatly to our next conundrum: what if a variable is used multiple times. In the example, this goes for x_1, something that we have ignored until now. The mathematical solution is simple: the partial derivative of a variable that is used multiple times, is just a summation of the adjoints arising from those uses. We see this in Equation 11, where $\bar{x}_1$ is calculated by adding the influence from r_1 and the influence from r_2 together. However, implementation-wise this can be a bit of a hurdle.

As mentioned, the trace is not in any order. This is unlike a typical Wengert list or tape. While assuring some order beforehand, or doing topological sort on the computational graph described by the trace, will in large part solve this problem, it also enforces linear execution of the reverse pass. And while it is not something we will linger on for now, allowing for concurrency or task parallelism while calculating the derivative might be a nice for a performance boost, and complement the inherent data-parallelism opportunities of array operations. So, to solve this, we want to include some form of reference counting. During the forward pass we could count how many times each variable is used in the trace. Since we need to store intermediate values anyway, keeping a counter for each of these variables seems like little extra work. Now, on the reverse pass we can check these reference counters and every time we find part of the adjoint for a variable, we decrement its associated counter. If a counter has not reached zero after we have decremented it, we know its adjoint is not yet complete, and we can ignore it for now. If it has, we can add up all the parts of the adjoint and continue from there. This is actually very similar to Kahn's algorithm for topological sorting[49], except that rather than sorting the graph beforehand, we immediately process the nodes as they become available (have all their incoming adjoints). This means we actually do execute the reverse pass in topological order, but by discovering this order as we go it allows us to not strictly do the reverse pass sequentially, something of which we will discuss the merits of further on. Provided there is only one output to the program, we know that all reference counters will eventually reach zero, and therefore we are assured we will calculate all adjoints. However, this provision is not as clear-cut as it seems. Currently, our DSL does not really have any room for multiple outputs, and as it is functional does not support any side effects. Instead, to provide multiple outputs, currently the only way is to output an array. If we keep arrays in the trace, an array as output would still count as a single value. There is a slight discrepancy between the trace and the output if we trace away arrays however: the program will still output an array, but only its individual items are present in the trace. This is not really a big problem, since the name of these individual outputs are derived from the name of the full array, but also because it would make little sense to trace away arrays from a program that outputs an array.

So, we find that our trace needs to be extended with two additional things in the forward pass: intermediate values and reference counters. We do this in Listing 22, in the data type `Forward`. Also, we introduce a clone of the `Traced` data type as `Forwarded`, as we need to reference the new `Forward` type in the constructors for maps and vectorized maps. We also replace the list structure of `Trace` with a key-value map. This is not strictly necessary, but it allows us to more quickly access the values in the map, while also clearly communicating there is no pre-set order to the trace. Each value in a `Forward` map is a 3-tuple consisting of respectively: the intermediate value, the traced operation performed, and the reference counter for this variable. Other than the added reference counting, and saving of intermediate values, the tracing process remains the same as it

was in Section 3.

```
data Forwarded
    = FLift TValue
    | FOp0   Op0
    | FOp1   Op1       String
    | FOp2   Op2       String String
    | FMap   [Forward] String
    | FMapV  Forward   String
    | FFold  Forward   String String
    | FFoldV Forward   String Forward String String

type Forward = Map String (TValue, Forwarded, Int)
```

Listing 22: Forward pass data structures

4.1 The Reverse Pass

As discussed, to facilitate our reverse pass we need both the reference counting and intermediate values. Now let us define a function `reverse` that does the reverse pass. This reverse pass should find all the adjoints in the program. So, it should take in an object of the `Forward` type and output a map containing the adjoints. In Listing 23 we define three constructors for adjoints: one for arrays, one for sparse arrays (represented by a single index and the associated value), and one for real values. We also define the `Reverse` type, which will contain these adjoints, and which is returned at the end of the reverse pass. The `Reverse` type maps the names of each part of the calculation to a 2-tuple containing a list of contributions of other adjoints, and its own final adjoint which uses maybe to indicate whether it has been calculated yet.

```
data Adjoint
    = AArray  [Float]
    | AReal   Float
    | ASparse Int Float

type Reverse = Map String ([Adjoint], Maybe Adjoint)
```

Listing 23: Definition of the `Adjoint` type

Now before going into precise implementation details we should look at the general picture once more. The forward pass provides us with three important components: the final output value of the forward evaluation, the trace on which to do our reverse pass, and the intermediate values we will need to actually calculate everything in the reverse pass. First off, the final output value is not actually important for the trace, were it not that it also stores its name in the constructor (for `TValue`, see Listings 4 and 8). This name points us where to start with the reverse pass, namely the step that produced this output value.

With our starting point clear, we can now start the reverse pass. The programmer will provide some sort of adjoint value (either a real number or an array of them, depending on the output of the regular program), which we will immediately assign to our output value in the reverse pass. This makes us ready to actually perform the rest of the reverse pass.

For any point in the reverse pass the process becomes simple. Given some "current" point in the computational graph, we look up the adjoint (which should have been established by now) and the forward trace item for this point. If the operation in the trace for the current point uses no other values (i.e. has no "ancestors"), we are done here and return the reverse mapping that contains all the adjoints we have found. If the operation does have ancestors, we look at the operation itself to determine how to transform the current point's adjoint for its ancestors. Then, if this transformation requires any intermediate values, we can look them up in the forward pass. Given the transformed adjoints, we assign these to the adjoint accumulation list for each ancestor. We also check if this list now has enough adjoints to match the reference counter in the forward pass. If it does not, we are done and can return the reverse mapping. However, if it does, we add up all the partial adjoints in the list together into the final adjoint and place it in the reverse mapping. Then finally, we start the same process for each ancestor of the current node that has its complete adjoint ready.

Now, let us talk implementation. We define two reverse pass functions, `reverse` and `resolve`, of which the first will only be a wrapper for the final value and its adjoint to be inserted, and the latter will actually perform the reverse pass. Both are shown in Listing 24. Listing 24 also introduces two helper functions: `combineAdjoints` for adding partial adjoints together into the final adjoint of a step in the computational graph, and `assignAdjoints` for transforming and assigning the adjoints to the ancestors of the current node. We will save the intricate details of `combineAdjoints` and `assignAdjoints` for later. Finally, we use the `explore` helper function to try and resolve all the ancestors of the current node as well.

We represent this process on the example program from Figure 1, in Figure 2. In Figure 2 we see the computational graph (forward pass) from Figure 1 first, as (A).

Then, with $f = r_2$ as our output value from the forward pass, we can call `reverse` with some adjoint a, the name of r_2, and the forward pass we just found to get the reverse graph at (B); this assigns a value to r_2 in the reverse map, which we call r_2' in Figure 2.

As part of the reverse map, r_2' contains two items: a list of partial adjoints (currently only containing a), and a final adjoint that has not been calculated yet (so is stored as a `Nothing`).

Now we can get into the main loop by calling `resolve` for r_2 with the reverse mapping created by `reverse`. As we can see in graph (C), as we find the partial adjoints r_2' and find that the reference counter in the forward pass is $1 = |r_2'|$, we can also call `combineAdjoints`, which transforms the list of partial adjoints in r_2' to the complete adjoint a.

Then we call `assignAdjoints` to get graph (D). `assignAdjoints` uses the forward pass to find the ancestors of the current node, in this case x_1 and r_1, and it also adds the final adjoint of the current node (r_2''s final adjoint is a) to their lists of partial adjoints.

Then `resolve` on r_2 calls `explore` leading to a `resolve` call to each of r_2's ancestors, with first up x_1 in graph (E). However, as we can see from the forward graph (A), we still lack the adjoint from r_1 to x_1 (highlighted with the red arrow in graph (E)), so we can not resolve x_1 yet.

We leave x_1 for later, and this leads us to the resolve call on r_1 in graph (F). Here we find that r_1 does have all its partial adjoints, so we call `combineAdjoints` to find that

```haskell
reverse :: String -> Adjoint -> Forward -> Reverse
reverse s a f = resolve s f $ Map.singleton s ([a], Nothing)

resolve :: String -> Forward -> Reverse -> Reverse
resolve s f r = case Map.lookup s r of
    -- Lookup the state of the provided name in the reverse mapping
    -- If its present, but its final adjoint not calculated, we need to check
    -- if we can calculate it
    Just (as, Nothing) -> case Map.lookup s f of
        -- Find the step taken and the reference counter in the forward pass
        -- If it is present, check whether or not the adjoint array contains
        -- items equal to the reference counter (so we know it is all there).
        Just (fd, c, _) -> if   length as >= c
                           -- If all partials are present, make the complete
                           -- adjoint and update the reverse map
                           then let (a,  r1) = combineAdjoints s f r
                                -- Then transform and assign the adjoints to
                                -- the ancestors of this current node.
                                    (r2, sa) = assignAdjoints fd s a f r1
                                -- Then try all the ancestors as well
                                in  explore sa r2
                           -- If we are not ready, just return the current
                           -- reverse map
                           else r
        -- If the named variable isn't present in the forward trace, we've
        -- got a problem, so we throw an error
        Nothing         -> error "Variable not in forward trace"
    -- If the adjoint is present, and its final adjoint is already calculated
    -- then we must already be done here, so just return the current reverse
    -- mapping
    Just _              -> r
    -- If the adjoint is missing from the reverse mapping entirely, it is because
    -- we haven't run into it at all yet, so we can also return the reverse mapping
    -- as it is.
    Nothing             -> r
    where
        -- Applies resolve over a list of ancestors
        explore :: [String] -> Reverse -> Reverse
        explore []      r' = r'
        explore (s':ss) r' = explore ss (resolve s' f r')
```

Listing 24: Definition of the reverse pass functions

the completed adjoint for r_1 is also a.

Again, with this adjoint found, we can now call `assignAdjoints` to bring the adjoint of r_1 to its ancestors x_1 and x_2. We will go into how later, but `assignAdjoints` looks in the forward pass to find that r_1 is a multiplication and appropriately transforms r_1's adjoint of a into $a \cdot x_2$ for x_1', and $a \cdot x_1$ for x_2'.

Now in our final steps represented in graph (H), we explore the ancestors of r_1, starting with x_1. We find that we now do have the correct number of partial adjoints for x_1, so we can add these together with `combineAdjoints` to get the final adjoint of x_1: $a+a \cdot x_2$. Now, after `combineAdjoints` we call `assignAdjoints` on x_1, however we will find that x_1 has no ancestors, which means that x_1's adjoint does not need to be assigned to anything and also that there is no further nodes to explore from x_1.

This means we move back to r_1's explore function, which leads us to x_2. Here we find again that we can use `combineAdjoints` to get x_2's final adjoint, and with `assignAdjoints` that x_2 has no further ancestors.

This then moves us back to the end of the explore function in r_1, which now finished, returns the updated reverse mapping to the end of the explore function of r_2, which also finishes, returning the updated reverse mapping to the original `reverse` function, and return our reverse pass to the programmer.

This gives us a global overview of how the reverse pass can be implemented, using the forward pass/trace we have discussed. Now there are a couple of questions that remain:

- Can we always add adjoints together?
- How do different operations differentiate?
- How do we maintain data-parallelism in array operations?
- How do we implement task-parallelism on the reverse pass?

In the following subsections we will get to all these questions.

4.2 Combining Adjoints

To go from a list of partial adjoints to a single combined adjoint is not quite as trivial as just adding all together. As mentioned before, in Listing 23, we defined three types of adjoints, an array, a sparse array, and a real number adjoint. The real number adjoint is not complicated, it is the default adjoint, and they can be freely added together. The difficulty comes in with the array adjoints. These adjoints are produced by operations on arrays. It will help us to realize now that the adjoint of an array with length n, will also have length n. In reality the adjoint array is no more than an array of adjoints for each item in the original array.

Knowing this we can start to discover how to add these adjoints together. Let us start by discussing adding the sparse and non-sparse array adjoints, as they are almost as simple as adding two real adjoints. Since we only add these together as partial adjoint of a single step in the computational graph, we know that the array adjoints whether sparse or not will always have the same length when added together. This means we can

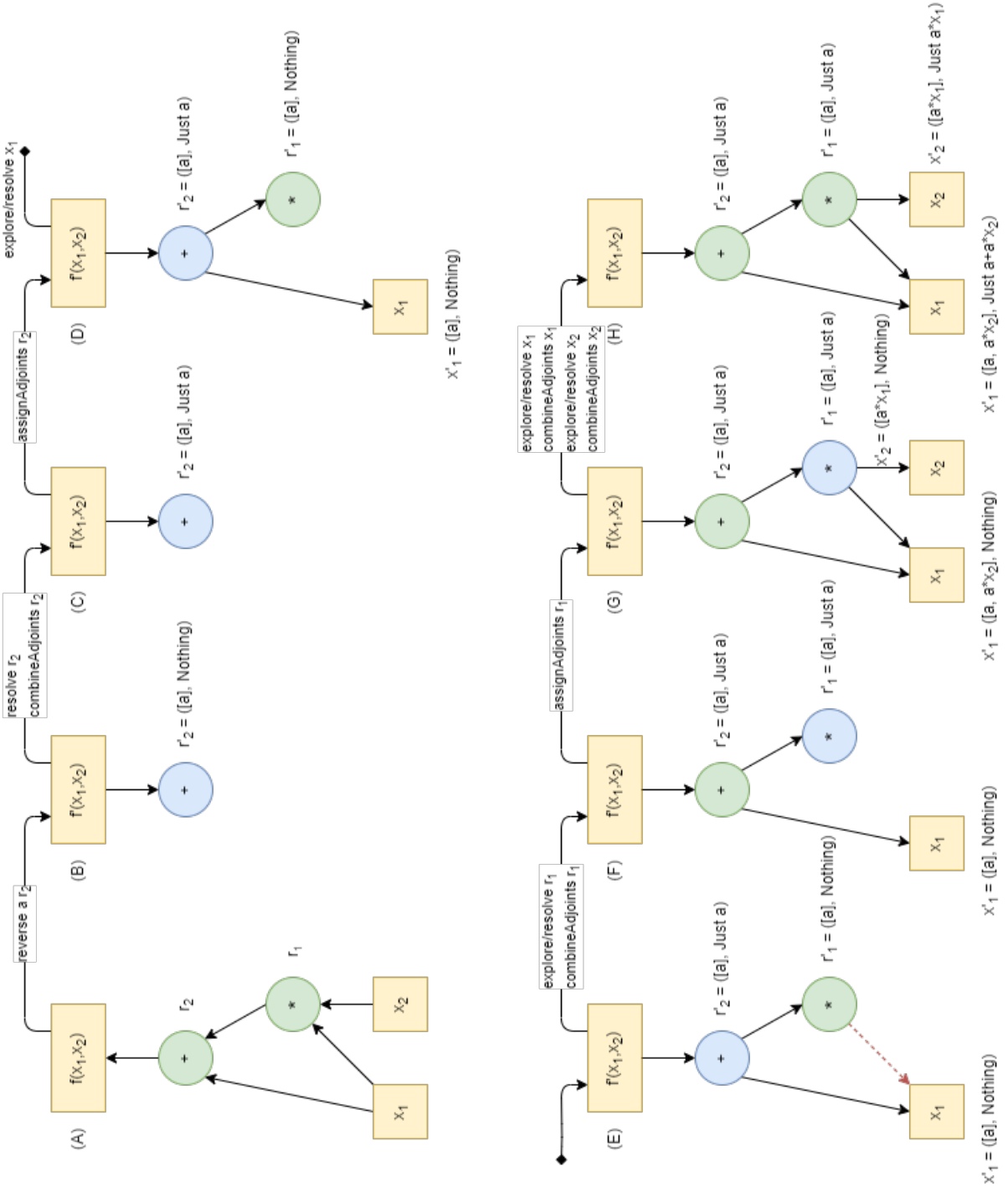

Fig. 2: Diagram representing the reverse pass process on the example from Figure 1.

just add these arrays of adjoints together element-wise, substituting 0 for all undefined items in the sparse array.

Only if we add two sparse arrays together, we might need to look up the length of the original array in the forward pass to know the length of these sparse arrays (or we could extend the sparse array constructor to allow for multiple defined items).

Now the most interesting part comes when we want to add an array adjoint to a real adjoint, or vice versa. Namely, there are two ways of doing this, which means that a binary adjoint addition operator is not associative. For example, say we find ourselves folding the values in some array a to some real value b. In the reverse pass, the adjoint of b will be a real adjoint, yet the adjoint of a will have to be an array adjoint. Now provided that a has some partial array adjoint (or is represented by an adjoint array filled with zeroes) we must think of a way to add b to the appropriate items. Now what these appropriate items in a are, and how b's adjoint needs to be transformed, are of course dependent on the function we fold over array a.

However, we will find that fold takes in an anonymous function, meaning that we need a special case for fold anyway. Only for our unary sum operator we perform a fold that is pre-programmed. We know that the sum's result is influenced once by each array item, and from the differentiation rules of addition we know that the adjoint of the sum's result is just moved to its ancestors without transformation or touching intermediate values. This means that we can just add the sum's adjoint to each array item.

The only other way to get from an array to a real value in our language is by using array indexing. However, this is very simple as well, as indexing does not change anything to the adjoint either. Since indexing refers to a specific item in the array, our partial adjoint for the array becomes a sparse array with the incoming adjoint on the indexed position.

We see the implementation of the helper function `combineAdjoints` in Figure 25, where we also define a binary adjoint addition operator ($<+$) to do most of the heavy lifting. It should be reiterated that the ($<+$) operation adding a real to an array only works because it will only be called by the sum operator.

```
(<+) :: Adjoint -> Adjoint -> Adjoint
(<+) (AArray as)   (AArray    bs) = AArray (zipWith (+) as bs)
-- Note: this works because it is only called by sum
(<+) (AArray as)   (AReal     bs) = AArray (map (+ b) as)
(<+) (AArray as)   (ASparse i bs) =
    let bs = drop i as
    in  AArray (take i as ++ (b + head bs) : tail bs)
(<+) (AReal  a)    (AArray    bs) = AReal (a + sum bs)
(<+) (AReal  a)    (AReal     b)  = AReal (a + b)
(<+) (AReal  a)    (ASparse _ b)  = AReal (a + b)
(<+) (ASparse i a) (AArray    bs) =
    let as = drop i bs
    in  AArray (take i bs ++ (a + head as) : tail as)
(<+) (ASparse _ _) (AReal     _)  = error "Cannot combine sparse and real
    adjoints, because the length of the sparse adjoint is unknown."
(<+) (ASparse _ _) (ASparse _ _)  = error "Cannot combine two sparse adjoints,
    because the length of the sparse adjoints are unknown."

combineAdjoints :: String -> Forward -> Reverse -> (Adjoint, Reverse)
combineAdjoints s f r =
    -- Get the partial adjoints and combine them together
    let (as, _) = r Map.! s
        a       = foldr (<+) empty as
    -- And add the completed adjoint to the reverse map
    in  (a, Map.insert s (as, Just a) r)
    where
        -- Provide an empty identity element
        empty :: Adjoint
        empty = case getValue s f of
            -- Check the intermediate value of s to reveal the right identity element
            (FArray _ xs) -> AArray (replicate (length xs) 0.0)
            (FReal  {})   -> AReal  0.0
            _             -> error "Type mismatch in combineAdjoints/empty"
```

Listing 25: Adjoint combination operator and adjoint summation function

4.3 Differentiating Operations

The actual differentiation rules are applied in the `assignAdjoints` function, where we take the complete adjoint of a node in the computational graph, transform it according to the operation performed in that node, and then add it as a partial adjoint to its ancestors. It is especially the transformation that means we need to write different differentiation rules for almost every operation.

Let us start with the easy part, the unary and binary mathematical operators. In our DSL these are: addition, multiplication, subtraction, and sine. Recall that the rules for addition and subtraction are similar, they just transform homomorphically, which means that any adjoint is just "passed" to their ancestors without any additional transformation. See the examples in Equation 14.

$$\begin{aligned} \frac{d(x+y)}{dz} &= \frac{dx}{dz} + \frac{dy}{dz} \\ \frac{d(x-y)}{dz} &= \frac{dx}{dz} - \frac{dy}{dz} \end{aligned} \tag{14}$$

Multiplication and sine are slightly more complicated, both requiring some intermediate value to compute the derivative. We have gone over multiplication before, we multiply the incoming adjoint with intermediate value of the other side of the multiplication. See the example in Equation 15.

$$\frac{d(x \cdot y)}{dz} = y \cdot \frac{dx}{dz} + x \cdot \frac{dy}{dz} \tag{15}$$

The derivative of a sine operation is the cosine on the intermediate value, see Equation 16.

$$\frac{d(\sin x)}{dy} = \cos x \cdot \frac{dx}{dy} \tag{16}$$

These are the mathematical operations that we may encounter in the trace. Recall that we traced away Booleans, so we do not have to worry about comparison operators. With these derivatives cleared up we can now program them into `assignAdjoints`. It is also useful to remember that `assignAdjoints` should return both the updated reverse mapping, and a list containing the ancestors of the provided node. We see the implementation in Listing 26. This uses another helper function called `addAdjoint`, which simply just adds the calculated partial adjoint to the list of partial adjoints of the relevant ancestor in the reverse mapping.

Another adjoint we should quickly cover is that of array indexing. As mentioned before, nothing happens to an intermediate value when it is indexed, so its adjoint will not be transformed either. While obvious, the adjoint of the indexed value should only be added to the that specific index in the array's adjoint. This is where our sparse adjoint comes in, where we represent a single item in the array without storing anything extra. We can see how it is used by `assignAdjoints` for indexing in Listing 27.

```
assignAdjoints :: Forwarded -> String -> Adjoint -> Forward -> Reverse
    -> (Reverse, [String])
-- Adjoint function for all unary operators (now only showing sine)
assignAdjoints (FOp1 op s1) _ a f r = case (op, a) of
    (Sin, AReal a') -> case getValue s1 f of
        -- Take the intermediate value and assign the adjoint to s1, also return
        -- the list of ancestors: s1
        (FReal _ x) -> (addAdjoint s1 (AReal $ a' * cos x) r, [s1])
        _           -> error "Type mismatch in assignAdjoints/FOp1/Sin"
    _               -> error "Type mismatch in assignAdjoints/FOp1"

-- Adjoint function for all binary operators
assignAdjoints (FOp2 op s1 s2) _ a f r = case (op, a) of
    -- Just add the adjoint to both ancestors
    (Add, _)        -> (addAdjoint s1 a (addAdjoint s2 a r), [s1, s2])
    -- For multiplication, get the intermediate values first
    -- we also deconstruct the adjoint, knowing it is a real number because
    -- (sin x) would return a real number.
    (Mul, AReal a') -> case (getValue s1 f, getValue s2 f) of
        (FReal _ v1, FReal _ v2) ->
            (addAdjoint s1 (AReal (a' * v2)) (addAdjoint s2 (AReal a' * v1) r),
            [s1, s2])
        _                        ->
            error "Type mismatch in assignAdjoints/FOp2/Mul"
    -- Similar to addition, only s2 gets a negative adjoint
    (Sub, AReal a') -> (addAdjoint s1 a (addAdjoint s2 (AReal -a') r), [s1, s2])
    _               -> error "Type mismatch in assignAdjoints/FOp2"
```

Listing 26: Defining `assignAdjoints` for sine, addition, subtraction, and multiplication.

```
assignAdjoints :: Forwarded -> String -> Adjoint -> Forward -> Reverse
    -> (Reverse, [String])
assignAdjoints (FOp1 op s1) _ a f r = case (op, a) of
    -- Add the sparse array and return the name of the array as ancestor
    (Idx i, AReal a') -> (addAdjoint s1 (ASparse i a') r, [s1])
    ...
...
```

Listing 27: Defining `assignAdjoints` for the indexing operation

4.3.1 The reverse pass on map operations and function closures

Unsurprisingly, the reverse pass is slightly more in-depth on array operations, like map. For a regular non-vectorized map, this is still fairly easy to wrap our heads around. Especially if we remember that our forward-pass provides us with the following constructor for such maps: FMap [Forward] **String**. Here we store every application of the mapped function to our array as its own sub-trace. Now it becomes easy to realize that the simplest way to reverse-pass over this mapping is just to call the `reverse` function on each of these forward sub-traces. This then provides us immediately with the adjoint of each original array item, which we can combine into an array adjoint for the original array. We give this array adjoint as a partial adjoint to the original array (as it is the ancestor of the mapping operation). It should be explicitly stated that mapping the `reverse` function reveals the derivative of a map: another map, but in reverse.

However, there is a slight obstacle yet to overcome; to do with the mapped function. What do we do if the function that is mapped over the array uses variables from anywhere outside the actual array? Recall that the functions that are mapped are lambda expressions that have access to any variables in the environment at the function's definition. As an example, we can see this expressed in Figure 3, where some mapped operation on an array $[b]$ uses a variable a to produce the array $[c]$. While the partial adjoints for this variable are computed during the reverse pass over every item, they are only stored in the reverse mapping for the particular item. And if we extract only the adjoint for the original array item from this reverse mapping, we would throw away these partial adjoints, making it possibly impossible for us to calculate the full reverse-pass of the program, as we will not be able to calculate the adjoint for these outside variables. We find that these variables, like a in 3 are "unofficial ancestors" of the mapping operation. They are used by the mapping operations, but can be a little hard to find, as they are hidden in the sub-traces.

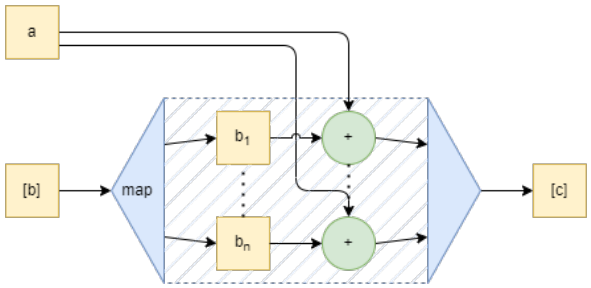

Fig. 3: Example of an unofficial ancestor to an array operation

Luckily for us the main difficulty with this problem is noticing it. Now we know that we have to extract this data into the main reverse pass, we can simply merge the item's reverse pass with the main one. We only need to be wary of extracting the partial adjoints for the original map, and combining them together, so we do not over count the number of partial adjoints against the reference counter of the original array (that in the forward pass only got increased by one for use in the map operation.) We see the whole process of finding the adjoint of a map in Listing 28.

Now with the non-vectorized map taken care of, we can move on to the vectorized map. While the idea of finding the adjoint to the map remains the same of course, we now find ourselves with a new problem: a lack of intermediate values. Recall that when tracing a

```
assignAdjoints :: Forwarded -> String -> Adjoint -> Forward -> Reverse
    -> (Reverse, [String])
...
assignAdjoints (FMap fss s1) s a _ r =
    let (as, r', ss) = reverseMap fss 0
        -- Fold sparse adjoints into single array adjoint
        a' = foldl (<+) (AArray $ replicate (length fss) 0.0) as
    in  (addAdjoint s1 a' r', Set.toList ss)
    where
        reverseMap :: [Forward] -> Int -> ([Adjoint], Reverse, Set String)
        reverseMap []     _ = ([], r, Set.empty)
        reverseMap (f:fs) i =
            let s'  = s ++ '!' : show i
                rx  = reverse f s' (indexAdjoint i a)
                -- Extract the array item for
                ax  = toSparse i $ fst $ combineAdjoints s' f rx
                -- Remove items from the original and destination array from this reverse pass
                -- and add the array items partial adjoint
                rx' = Map.delete s' (Map.delete (s1 ++ '!' : show i) rx)
                -- Find the results of the rest of the map
                (axs, rxs, sxs) = reverseMap fs (i + 1)
            in  (
                -- Add this sparse partial to the list
                ax : axs,
                -- Add rx' to the main reverse pass
                Map.unionWith unionReverse rxs rx',
                -- Add relevant keys to the set of ancestors
                Set.union sxs $ Map.keysSet rx'
            )
        toSparse :: Int -> Adjoint -> Adjoint
        toSparse i (AReal a') = ASparse i a'
        toSparse _ _          =
            error "Type mismatch in assignAdjoints/FMap/toSparse"
```

Listing 28: Implementation of `assignAdjoints` for the map operation

vectorized map, we only stored the trace of a single item in the array. This was possible because, as we had found the mapped function would not branch, the trace would be the same for each item. Now, provided that we did store all intermediate values for this mapping operation, we can perform the same reverse map for the vectorized trace, as with the nonvectorized trace.

On map parallelism

It should be clear that there is a major inherent parallelism opportunity for each of the mapping operations. For the vectorized map this is very clear: data parallelism. This is true in evaluation, and is still true in the reverse-pass. Namely, as the reverse pass is the same for all items, we can use a data parallel mapping function to run the reverse pass in a vectorized manner as well.

For the non-vectorized map we cannot be sure the operations are the same for each item. This means that data parallelism is off the table, however we can still use task parallelism. This would not be hard to implement as each item has its own reverse call which can easily be run task parallel. Of course, one would need to exercise some caution with the collection of the partial adjoints and reverse maps, but this is not a new problem. The combination of the partial adjoints would still need to happen sequentially.

We will talk more about parallelism when we talk about the reverse map on folds.

4.3.2 The reverse pass on fold operations

The regular fold operator in our language (for branching functions) is simply implemented as a sequential left-to-right fold. This is mainly because we can not reliably apply data parallelism on a function that may behave differently for different elements. However, this makes the reverse-pass over such a fold extremely easy, since this operation can be captured in a single forward sub-trace. Like what we saw with maps, we can just run our general `reverse` function over this sub-trace, and find the adjoint for the original array, and the identity element. Also like with maps, we can then combine this reverse mapping with the main reverse mapping to make sure we do not lose out of any adjoints for informal ancestors that may arise from the closure on the folding function. We see this put into code in Listing 29.

```
1  assignAdjoints :: Forwarded -> String -> Adjoint -> Forward -> Reverse
2      -> (Reverse, [String])
3  ...
4  assignAdjoints (FFold f s1 s2) s a _ r =
5          -- Get the reverse mapping
6      let rf = reverse f s a
7          -- Extract the adjoint for the array and the identity elements
8          aa = fromJust $ snd $ rf Map.! s1
9          az = fromJust $ snd $ rf Map.! s2
10         -- Combine the reverse map with the main reverse map
11         r' = Map.unionWith unionReverse r (Map.delete s1 (Map.delete s2 rf))
12     in  (addAdjoint s1 aa (addAdjoint s2 az), [s1, s2])
```

Listing 29: Reverse pass over a sequential fold

As discussed, our vectorized fold operation uses a segmented fold followed by a sequential fold. Of course if we have more threads available than we have items in our array, we use a sequential fold instead. Since in the reverse-pass we pass over the second step before the first step, it is important to recall that to properly gather the results from the first step sub-traces, we use an `FJoin` constructor, which we can use in the reverse pass to pass the adjoints back to the correct sub-trace. We can see this all come together in the implementation in Listing 30.

Of course, when we encounter the `FJoin` constructor in the reverse pass, we still need to apply the reverse-pass on it. However, since the `FJoin` is not a real numerical operation, it also has no derivative. So we just ignore `FJoin`, as it is the starting point for the second step anyway, it has no values depending on its derivative. The derivative it does get is an array containing derivatives for the joined values (from the first step), to be used in the reverse-pass for the first step.

On the sum operation

The DSL implements a special kind of fold as a unary operator, namely the sum operation. Recalling that the sum operator just adds up the values in an array, we can implement this as a fold over the array, using the addition operator and 0 as the identity element. However, the reverse pass over a sum operation is very simple. Starting with some real adjoint a at the result of the sum, this adjoint is then passed through all the addition operators to all the members of the original array. This allows us to quickly make the adjoint for the original array, namely an array adjoint where all values are a. Taking such a shortcut also means leaving out the adjoints for all intermediate calculations, however since we intuitively know what they are for sum, this should not really pose a problem. In Listing 31 we see the implementation of the `assignAdjoints` function for the sum operation.

The main takeaway here is not really about sum, however. It is that special cases exist in the reverse pass, even if they seem somewhat generic in the regular program. Finding these special cases can increase the speed of the reverse pass. While the sum operation might not be too exciting an example, it highlights that we should be on the lookout for these special cases, especially for the more complex ones (as they will profit more from the potential speed-up.)

On fold parallelism

Fold parallelism is also a little more complicated than map parallelism. An easy way to think about why, is by visualizing the flow of data in both operations. In map, we start with an array of n items, and end with one. Every array element "flows" from some index in our original array to the same index in the resulting array, never intersecting with any of the other elements. With folding operations, it is especially this intersection of the dataflows that makes it hard to parallelize in the reverse pass. If we imagine the values in the original array as being funnelled into the final result, we quickly realize that there is no operation that does the opposite: there is no "unfold" operation. However, both the individual segments in the first step of the fold, and the combination in the second step are relatively simple sequential folds. While the intermediate values in a folding operation are ignored in regular use, we actually store them as part of our forward traces. This means that our sequential folds are more like a scan, a left-to-right scan to be precise. Realizing this, we can imagine our reverse-pass as a right-to-left scan, applying the fold (or calculating its adjoints) in reverse if you will. Of course this works

```
assignAdjoints :: Forwarded -> String -> Adjoint -> Forward -> Reverse
    -> (Reverse, [String])
...
assignAdjoints (FFoldV f1 q f2 s1 s2) s a _ r =
    if   Map.null f2
    -- If the forward pass for step 2 is empty, we used a sequential fold
    then let rf = reverse f1 s a
             -- Extract the adjoint for the array and the identity elements
             aa = fromJust $ snd $ rf Map.! s1
             az = fromJust $ snd $ rf Map.! s2
             -- Combine the reverse map with the main reverse map
             r' = Map.unionWith unionReverse r
                (Map.delete s1 (Map.delete s2 rf))
         in  (addAdjoint s1 aa (addAdjoint s2 az), [s1, s2])
    -- If not we are dealing with a proper vectorized fold
             -- Reverse pass over the second step first
    else let r2 = reverse f2 s a
             -- Extract the adjoint for the join from step 2
             a2 = fromJust $ snd $ r2 Map.! q
             -- Combine the reverse map for step 2 with the main one
             r' = Map.unionWith unionReverse r (Map.delete q r2)
             -- Get the join's contents
             ss = getJoin
             -- Get the update reverse mapping, the partial adjoints of the array,
             -- and the adjoint of the identity element.
             (r'', a1s, z') = getParts ss (fromJust a2) r'
             -- Combine the partial adjoints for the array into one
             a1 = foldl (<+) (AArray $ replicate (Map.size f1) 0.0) a1s
         in  (addAdjoint s1 a1 (addAdjoint s2 z' r''), [s1, s2])
    where
        getJoin :: [String]
        getJoin = case f2 Map.! q of
            (FJoin ss, _, _) -> ss
            _                -> error "Type mismatch in assignAdjoints/FFoldV"

        -- Get the updated reverse mapping, the adjoint for the array,
        -- and the adjoint for the identity element
        getParts :: [String] -> Adjoint -> Reverse
            -> (Reverse, [Adjoint], Adjoint)
        getParts []     _               rv = (rv, [], AReal 0.0)
        getParts (s':ss) (AArray (a':as)) rv =
            let rx = reverse f1 s' (AReal a')
                ax = fromJust $ snd $ rx Map.! s'
                (rxs, axs, zxs) = getParts
                rx' = Map.unionWith rxs (Map.delete s1 (Map.delete s2 rx))
            in  case Map.lookup s2 rx of
                -- Update the adjoint for the identity element only if it was used
                -- by this segment.
                Just (zs, _) -> (rx', ax : axs, foldl (<+) zxs zs)
                Nothing      -> (rx', ax : axs, zxs)
```

Listing 30: Reverse pass over a vectorized fold

```
assignAdjoints :: Forwarded -> String -> Adjoint -> Forward -> Reverse
    -> (Reverse, [String])
...
assignAdjoints (FOp1 op s1) _ a f r = case (op, a) of
    ...
                        -- We need to get the arrays length for the adjoint
    (Sum, AReal a') -> case getValue s1 f of
        (FArray _ xs) ->
            (addAdjoint s1 (AArray $ replicate (length xs) a') r, [s1])
        _             -> error "Type mismatch in assignAdjoints/FOp1/Sum"
...
```

Listing 31: Implementation of the sum operator in `assignAdjoints`

on sequential fold, but it works on the data-parallel fold as well, and we can maintain almost the same parallelism as before. First, we calculate the adjoints in the second step using a sequential right-to-left scan, and then we do the same for the segments in the first step, now in parallel. Of course, the resulting adjoint arrays of each segment would still need to be combined sequentially, which is not necessary in the regular fold. Still, this preserves the data-parallelism present in the fold, which was our goal.

4.4 Task parallelism in the reverse pass

We have paid some attention to data parallelism for data parallel operations, namely map and fold. However, we can also more generally implement task parallelism on the reverse pass as a whole. This is mainly why we have previously been adamant about not sorting the array, and having unique but non-sequential names (they may in fact be sequential, but should not be treated as such): it means we can execute the reverse pass in any order as long as we make sure to keep an eye out for missing partial adjoints. Again, we compare the number of collected partial adjoints of a computational step, to the number of references made to that step (according to the forward pass), and we only combine the adjoints if they match. This way we know we do not calculate the step's adjoint prematurely, but we also know that if we cannot calculate the adjoint, another part of the reverse pass will, as it will provide the last piece of that adjoint.

Eventually this comes together in the fact that we may perform the reverse pass completely asynchronously. We start with one thread at the end of the computation, but whenever part of the computation assigns adjoints to multiple ancestors, we can check these ancestors using task parallelism. We can just spawn new threads to deal with all but one ancestor, and continue the original thread on the remaining ancestor, and combine results back into the final result later. For large or complex traces this might actually allow us to significantly speed up our reverse pass.

It should be noted that our current implementation does not quite allow for this kind of task parallelism, because right now it returns the reverse mapping back to the starting point, meaning that the through-putting of the reverse mapping from one ancestor to the next forces sequential execution. However, this can easily be fixed using a mutable variation of the reverse mapping. While this would violate purity (which does not allow for mutable variables), the speed-up might be worth it.

4.5 Finding Complexity

With now the reverse pass figured out, we can start talking about estimating a time complexity for this program. We will try to do so with regard to the execution time of the regular program, which we will denote with n, where n is a stand-in for the number of steps in our execution graph. This means that regardless of the functions actual complexity, for this section the complexity of executing a function consisting of n-substeps, would have a complexity of $O(n)$.

For the forward pass we find that any recording of references to reference counters, or storing intermediate values should only provide us with a small overhead on the regular execution of the program. After all, we only count direct references, and we already have to calculate the intermediate values for execution anyway, storing them is not much of a problem. Of course, we do gain some time complexity overhead depending on the storage method. For tracing we used a built-in Haskell list, which can store new items in $O(1)$ time. However, for ease of use, we switched to a map (associative array) when we expanded to the forward pass. While of course lookup and getting in a map of size m is quicker than in an list of size m ($O(\log m)$ as opposed to $O(m)$), insertion on a map also takes $O(\log m)$ time. This means that we can assume our forward pass roughly takes $O(n \log n)$ time: for n steps in the execution, we record intermediate values at the cost of $O(\log n)$ time

Of course, this overhead in the forward pass, allows us to safe a little time in the reverse pass. Let us assume that for each step in our reverse pass, we need to retrieve the intermediate value at a cost of $O(\log n)$. Furthermore, we also need to write a partial adjoint to each ancestor of the step. Since we store these (partial) adjoints in a map as well, we know this will take $O(\log n)$ time to store for each ancestor. Luckily for us, each operation has only at most three ancestors, so we can still keep this overhead at $O(\log n)$.

Unfortunately, not all can currently be covered in a $O(\log n)$ overhead, because of the array operations. After finishing a map or fold operation, we need to join the reverse passes over the sub-traces back into the main trace. This union operation takes $O(m)$ time, where m is the size of the smaller map. There are two ways to remedy this however. First, we could make the reverse mapping a mutable variable. This way we can write new adjoints to the main reverse mapping directly, meaning we do not need to do any union operations. In a similar vein we could write a special reverse function for sub traces like this, that produce a new reverse mapping based on what happened in the sub-traces, while keeping the main trace up until that point intact. This means we can maintain purity, however it also means that we can not parallelize, as every sub-trace would need the reverse mapping of the previous sub-trace. (Or we could union these sub-trace reverse mappings again, but that would bring us back to the problem of union's time complexity.)

Regardless of how we wish to solve it, the current time complexity of the overhead produced by these operations is $O(n)$, meaning the time complexity of the reverse map becomes $O(n^2)$. When allowing mutability on the reverse mapping we could bring this down to an $O(\log n)$ overhead, or a time complexity of $O(n \log n)$.

This brings the total time complexity of the whole AD process as described to $O(n^2 + n \log n) \approx O(n^2)$, which is not great. However, we can also see that this overhead occurs due to implementation details. For instance, in general we would describe the

insertion and lookup complexity of an associative array (with size m) as $O(1)$ rather than $O(\log m)$, which eliminates these logarithmic overheads from the run altogether. However, due to implementation details in the mapping (provided by the containers package in Haskell), this turns out to be $O(\log m)$. Also, for the unions, we claim an $O(n)$ overhead, which is truly the worst case, assuming that all adjoints are already present in the reverse mapping (but we somehow need to add some more). Generally (and ideally), the reverse mappings for these sub-traces would be much smaller than those of the main reverse mapping, meaning a smaller overhead as well. If we assume the largest of these sub-trace reverse mappings has some size $k \ll n$, and we use the regular insert/lookup time complexity of $O(1)$, we find an idealized time complexity of $O(nk+n)$, which is much closer to the optimal time complexity $O(n)$ (as the complexity of the reverse-pass is ideally upper-bounded by $6 \cdot n$).

Finally, the speed-up we expect from task and data parallelism is wholly dependent on the shape of the computational graph, and the number of threads we can run simultaneously. For task parallelism, the ideal shape of the computational graph would be a tree (with the final result as the root), because it will allow us to make use of as much task parallelism as possible, without running into dead ends (due to missing partial adjoints). Similarly, data parallelism provides great advantage for vectorizable operations on large arrays. Of course, this dependency on the shape of the program means we cannot make a general statement about how these speed-ups would actually translate to time complexity.

It should be noted though, that especially data-parallelism has been left by the wayside by the large AD libraries, in favour of machine parallelism (where data and instructions are separated over multiple machines). While theoretically more scalable than regular data parallelism, machine parallelism is useful mostly for specific use cases like machine learning, and is also financially more costly. This means that machine parallelism is mostly reserved for large companies who can leverage multiple machines or even data centres to do this, while leaving groups with less resources out in the cold somewhat.

5 Conclusion

In this thesis, we set out to find a way to preserve data-parallelism through a reverse-mode automatic differentiation approach using tracing. We did so mainly by defining a domain-specific language (DSL) for Haskell, and implementing tracing and automatic differentiation on that DSL.

We found our first obstacle in the tracing, despite or maybe even because of its ubiquity, it was hard to find a proper definition that fit our use case. In fact, a formal definition of tracing seemed to be largely absent from literature, excluding some specialized definitions. So we first had to start by defining tracing, at least enough for our use case. We came to two insights here. First, we can decide on what and how to trace a program, by deciding what data types and data structures we wish to keep in the trace, and which we would rather remove or "trace away". Second, a uniform way of tracing is unlikely to exist, at least on a higher-level of programming. This would be due to many operations that may require special case implementations, some even dependent on the types we wish to keep in the trace. So while a completely formal universal definition of tracing was off the table, we still managed to create two logic assertions that, although quite broad still, would help us implement tracing on the DSL we wished to explore.

With the tracing worked out, we could now continue to the automatic differentiation. Reverse-mode AD exists of two phases, a forward pass and a reverse pass. The forward pass would be largely covered by tracing, however it needed a couple adjustment for it to be useful. First off, the trace was lacking intermediate values of the program, which would be needed for the reverse pass. Recalculating these values during the reverse pass would be very inefficient, so instead we modified the trace to also store intermediate values. The second problem was a little more tricky: while the trace would provide us with all the operations done, including a way to reverse pass through the trace by reference names, it did not provide the full structure of the computational graph. To be more precise: while calculating the adjoint for some node a on the computational graph, we could not be sure whether this was the only edge leading to a, or if there were others. This is problematic, because to continue the reverse pass from a to its ancestors, we would need to be sure the adjoint for a was calculated correctly. Of course, this could be easily solved by introducing some sorting algorithm to the trace before the reverse pass, however this would force sequential execution of the reverse pass. We wanted to avoid this as to leave room for task parallelism in the reverse pass: where we could introduce multiple sub-tasks from a node in the computational graph that has multiple ancestors. We managed to solve this problem by drawing inspiration from Kuhn's topological sort algorithm. First, we add reference counters to each node in the computational graph, for which we do the counting during the forward pass. Then on the reverse pass, when we have a calculated adjoint we wish to assign to an ancestor, we just add it to a list of partial adjoints for that ancestor. Now we can check the length of that list of partial adjoints to the reference counter associated with that node, and if they are equal, we would know we had collected all the partial adjoint and could calculate the complete adjoint for that node. Using this method, we can enforce an implied topological sort without enforcing that sort beforehand, which allows us the use of task parallelism.

Now using this carefully constructed forward pass, we would be able to do our reverse pass quite easily. However, we still had one problem left to deal with: closures. Since our DSL only allowed for lambda functions as closures, we already made sure to capture any relevant information in the environment on the forward pass. However, this meant that in the reverse pass our lambda functions could call on variables outside the function scope, variables that needed to be updated with relevant adjoints as well. For regularly applied functions this was no problem, after all the operations done by that applications would just appear in the trace, leaving no opaque closures for us to deal with. However, when we introduced arrays to our DSL, and array operations like map, generate, and fold, we ran into the opaqueness of these functions. While the function themselves where traced away into sub-traces for these array operations, these array operations themselves would not clearly display any variables outside the function scope as ancestors. When we would pass over them in the reverse pass, and use an independent reverse pass to calculate through the sub-traces, the adjoint contributions for variables irrelevant to the input array would be lost. Clearly this is unacceptable, because this would mean missing contributions, and incalculable adjoints for these informal ancestors. Luckily the main problem here was identifying it, as it could easily be remedied. We simply combine these reverse passes over the sub-traces with the main reverse pass, such that all contributions are saved. The main takeaways from the reverse pass implementation were the insight we gained into closure, and the reverse pass over data-parallel array operations. With our forward pass storing the nature of these array operations in sub-traces, we could leverage these sub-traces to do the reverse pass. We found that for the operations included in our DSL, we could leverage data-parallelism on the reverse pass in the same places data-parallelism was used in regular execution. Similarly, for any

point in the regular execution where we could implement task parallelism, we could also do so in the reverse pass.

With all this done, we ended up with a reverse-mode AD algorithm that would work on our entire DSL. While this DSL was not that impressive in scope, it did show us a clear way to use tracing for automatic differentiation. Furthermore, it showed us how we could maintain data parallelism in the reverse pass. We also found that the performance of our implementation left some room for improvement. However, by either letting go of purity for the reverse mapping and using more efficient data structures, we might be able to reach a more desirable time complexity.

5.1 Future work

Quite some work still remains. Such as an expansion to our tracing definition. This could either be an exploration into more complex type structures, or formalization of the definition we started. Especially with a formalized definition, one might be able to reach some more interesting correctness proofs than the two assertions we found. However, the question remains whether tracing can be formalized; we noted that we need special handling of a lot of cases, which might not be neatly formalizable.

Furthermore, we left the actual implementation of our AD algorithm at a high-level DSL, which is quite different from actual implementation on a lower level or GPU programming. While our findings should carry over regardless, it would be most interesting to see what kind of performance improvements can be gained by using data and task parallelism in the reverse pass for AD. Such an implementation should probably also include real parallelism, rather than parallelism hinted to by operations that could be parallelized.

In a similar vein, we probably would want to expand the operations of our DSL, to create more definitions for tracing and AD on specific operations. Plenty of parallel array operations remain, including scan, scatter, and gather to name just a few. There are also still types structures yet to explore, and types in general that may need to be traced (away).

Finally, but perhaps most importantly, there is the issue of higher-dimensional arrays. Our implementation in this paper dealt only with 1-dimensional arrays, which allowed us to sidestep complex situations, like multidimensional folds. While we can reasonably assume the theory laid out in this work should hold in higher dimensions as well, as there is nothing particularly special about higher-dimensional array operations, we could also imagine it becoming a difficult task to implement.

References

[1] Charles C Margossian. A review of automatic differentiation and its efficient implementation. *Wiley interdisciplinary reviews: data mining and knowledge discovery*, 9(4):e1305, 2019.

[2] Louis B Rall. The arithmetic of differentiation. *Mathematics Magazine*, 59(5):275–282, 1986.

[3] Seppo Linnainmaa. Taylor expansion of the accumulated rounding error. *BIT Numerical Mathematics*, 16(2):146–160, 1976.

[4] Andreas Griewank. Achieving logarithmic growth of temporal and spatial complexity in reverse automatic differentiation. *Optimization Methods and software*, 1(1):35–54, 1992.

[5] Christian H Bischof and H Martin Bücker. Computing derivatives of computer programs. Technical report, Argonne National Lab., IL (US), 2000.

[6] Claus Bendtsen and Ole Stauning. Fadbad, a flexible c++ package for automatic differentiation. Technical report, Technical Report IMM–REP–1996–17, Department of Mathematical Modelling . . . , 1996.

[7] Andreas Griewank, David Juedes, and Jean Utke. Algorithm 755: Adol-c: A package for the automatic differentiation of algorithms written in c/c++. *ACM Transactions on Mathematical Software (TOMS)*, 22(2):131–167, 1996.

[8] Pierre Aubert, Nicolas Di Césaré, and Olivier Pironneau. Automatic differentiation in c++ using expression templates and. application to a flow control problem. *Computing and Visualization in Science*, 3(4):197–208, 2001.

[9] Robin J Hogan. Fast reverse-mode automatic differentiation using expression templates in c++. *ACM Transactions on Mathematical Software (TOMS)*, 40(4):1–16, 2014.

[10] Laurent Hascoet and Valérie Pascual. The tapenade automatic differentiation tool: Principles, model, and specification. *ACM Transactions on Mathematical Software (TOMS)*, 39(3):1–43, 2013.

[11] Andreas Griewank and Andrea Walther. *Evaluating derivatives: principles and techniques of algorithmic differentiation*. SIAM, 2008.

[12] Fei Wang, Daniel Zheng, James Decker, Xilun Wu, Grégory M Essertel, and Tiark Rompf. Demystifying differentiable programming: Shift/reset the penultimate backpropagator. *Proceedings of the ACM on Programming Languages*, 3(ICFP):1–31, 2019.

[13] Faustyna Krawiec, Simon Peyton Jones, Neel Krishnaswami, Tom Ellis, Richard A Eisenberg, and Andrew Fitzgibbon. Provably correct, asymptotically efficient, higher-order reverse-mode automatic differentiation. *Proceedings of the ACM on Programming Languages*, 6(POPL):1–30, 2022.

[14] Matthijs Vákár and Tom Smeding. Chad: Combinatory homomorphic automatic differentiation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 44(3):1–49, 2022.

[15] Conal Elliott. The simple essence of automatic differentiation. *Proceedings of the ACM on Programming Languages*, 2(ICFP):1–29, 2018.

[16] Robert Schenck, Ola Rønning, Troels Henriksen, and Cosmin E Oancea. Ad for an array language with nested parallelism. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15. IEEE, 2022.

[17] Tom J Smeding and Matthijs IL Vákár. Efficient dual-numbers reverse ad via well-known program transformations. *Proceedings of the ACM on Programming Languages*, 7(POPL):1573–1600, 2023.

[18] Christian Bischof. *Issues in parallel automatic differentiation*. Argonne National Lab., 1991.

[19] Christian Bischof, Niels Guertler, Andreas Kowarz, and Andrea Walther. Parallel reverse mode automatic differentiation for openmp programs with adol-c. In *Advances in automatic differentiation*, pages 163–173. Springer, 2008.

[20] Dougal Maclaurin. *Modeling, inference and optimization with composable differentiable procedures*. PhD thesis, Harvard University, Graduate School of Arts & Sciences, 2016.

[21] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016.

[22] Roy Frostig, Matthew James Johnson, and Chris Leary. Compiling machine learning programs via high-level tracing. *Systems for Machine Learning*, 4(9), 2018.

[23] Bo Joel Svensson and Josef Svenningsson. Defunctionalizing push arrays. In *Proceedings of the 3rd ACM SIGPLAN Workshop on Functional High-performance Computing*, pages 43–52, 2014.

[24] Guy E Blelloch. *NESL: a nested data parallel language*. Carnegie Mellon Univ., 1992.

[25] Shail Aditya, Arvind, Jan-Willem Maessen, Lennart Augustsson, and Rishiyur S Nikhil. Semantics of ph: A parallel dialect of haskell. In *In Proceedings from the Haskell Workshop (at FPCA 95)*, pages 35–49, 1995.

[26] Jonathan Michael David Hill. *Data parallel lazy function programming*. PhD thesis, Queen Mary, University of London, 1995.

[27] Christoph A Herrmann and Christian Lengauer. Parallelization of divide-and-conquer by translation to nested loops. *Journal of functional programming*, 9(3):279–310, 1999.

[28] Nils Ellmenreich, Christian Lengauer, and Martin Griebl. Application of the polytope model to functional programs. In *Languages and Compilers for Parallel Computing: 12th International Workshop, LCPC'99 La Jolla, CA, USA, August 4–6, 1999 Proceedings 12*. Springer, 2000.

[29] Manuel MT Chakravarty, Gabriele Keller, Roman Lechtchinsky, and Wolf Pfannenstiel. Nepal—nested data parallelism in haskell. In *Euro-Par 2001 Parallel Processing: 7th International Euro-Par Conference Manchester, UK, August 28–31, 2001 Proceedings 7*, pages 524–534. Springer, 2001.

[30] Gabriele Cornelia Keller. *Transformation-based implementation of nested data parallelism for distributed memory machines*. PhD thesis, Technical University of Berlin, Germany, 1999.

[31] Gabriele Keller and Manuel MT Chakravarty. Flattening trees. In *Euro-Par'98 Parallel Processing: 4th International Euro-Par Conference Southampton, UK, September 1–4, 1998 Proceedings 4*, pages 709–719. Springer, 1998.

[32] Manuel MT Chakravarty and Gabriele Keller. More types for nested data parallel programming. *ACM SIGPLAN Notices*, 35(9):94–105, 2000.

[33] Manuel MT Chakravarty and Gabriele Keller. Functional array fusion. In *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming*, pages 205–216, 2001.

[34] Dror G Feitelson. Packing schemes for gang scheduling. In *Job Scheduling Strategies for Parallel Processing: IPPS'96 Workshop Honolulu, Hawaii, April 16, 1996 Proceedings 2*, pages 89–110. Springer, 1996.

[35] Gabriele Keller and Manuel MT Chakravarty. On the distributed implementation of aggregate data structures by program transformation. In *Parallel and Distributed Processing: 11th IPPS/SPDP'99 Workshops Held in Conjunction with the 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing San Juan, Puerto Rico, USA, April 12–16, 1999 Proceedings 13*, pages 108–122. Springer, 1999.

[36] Manuel MT Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, Gabriele Keller, and Simon Marlow. Data parallel haskell: a status report. In *Proceedings of the 2007 workshop on Declarative aspects of multicore programming*, pages 10–18, 2007.

[37] Simon Peyton Jones, Roman Leshchinskiy, Gabriele Keller, and Manuel MT Chakravarty. Harnessing the multicores: Nested data parallelism in haskell. In *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2008.

[38] Sven-Bodo Scholz. Single assignment c-functional programming using imperative style. In *Proceedings of IFL*, volume 94, 1994.

[39] Sven-Bodo Scholz. Single assignment c: efficient support for high-level array operations in a functional setting. *Journal of functional programming*, 13(6):1005–1059, 2003.

[40] Clemens Grelck and Sven-Bodo Scholz. Generic parallel array programming in sac. In *21. Workshop der GI-Fachgruppe 2.1.4 Programmiersprachen und Rechenkonzepte, Bad Honnef, Germany*, pages 43–53, 2005.

[41] Gabriele Keller, Manuel MT Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, and Ben Lippmeier. Regular, shape-polymorphic, parallel arrays in haskell. *ACM Sigplan Notices*, 45(9):261–272, 2010.

[42] C Barry Jay and J Robin B Cockett. Shapely types and shape polymorphism. In *Programming Languages and Systems—ESOP'94: 5th European Symposium on Programming Edinburg, UK, April 11–13, 1994 Proceedings 5*, pages 302–316. Springer, 1994.

[43] Manuel MT Chakravarty, Gabriele Keller, Sean Lee, Trevor L McDonell, and Vinod Grover. Accelerating haskell array codes with multicore gpus. In *Proceedings of the sixth workshop on Declarative aspects of multicore programming*, pages 3–14, 2011.

[44] Justin Slepak, Olin Shivers, and Panagiotis Manolios. An array-oriented language with static rank polymorphism. In *ESOP*, volume 8410, pages 27–46. Springer, 2014.

[45] Kenneth E Iverson. A programming language. In *Proceedings of the May 1-3, 1962, spring joint computer conference*, pages 345–351, 1962.

[46] Troels Henriksen, Niels GW Serup, Martin Elsman, Fritz Henglein, and Cosmin E Oancea. Futhark: purely functional gpu-programming with nested parallelism and in-place array updates. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 556–571, 2017.

[47] Dougal Maclaurin, Alexey Radul, Matthew J Johnson, and Dimitrios Vytiniotis. Dex: array programming with typed indices. In *Program Transformations for ML Workshop at NeurIPS 2019*, 2019.

[48] Adam Paszke, Daniel Johnson, David Duvenaud, Dimitrios Vytiniotis, Alexey Radul, Matthew Johnson, Jonathan Ragan-Kelley, and Dougal Maclaurin. Getting to the point. index sets and parallelism-preserving autodiff for pointful array programming. *arXiv preprint arXiv:2104.05372*, 2021.

[49] Arthur B Kahn. Topological sorting of large networks. *Communications of the ACM*, 5(11):558–562, 1962.

A ADT Evaluation

In Section 3.2, we introduced an extended lambda calculus. In this section we will quickly go over how an evaluator function for this ADT would look like in Haskell. We define our evaluator function in Listing 32, using the definitions of `Expression`, `Value`, and `Environment` from Listing 3.

```haskell
eval :: Environment -> Expression -> Value

eval n (EApply e1 e2) =
        -- Evaluate e1 and e2 first
    let v1 = eval n e1
        v2 = eval n e2
    in  case v1 of
            -- Only apply v1 to v2 if v1 is a function as expected
            VFunc f -> f v2
            _       -> error "Type mismatch in eval/EApply"

eval n (EIf e1 e2 e3) =
    -- Evaluate e1 as the condition of the if-then-else statement
    case eval n e1 of
        -- If e1 evaluates to true, evaluate e2
        VBool True  -> eval n e2
        -- Otherwise, evaluate e3
        VBool False -> eval n e3
        _           -> error "Type mismatch in eval/EIf"

-- For abstractions, we return the function by moving the evaluation into the body.
-- Where we insert the anonymous value x into the environment as it was when the
-- function was defined.
eval n (ELambda s1 e1) = VFunc $ \x -> eval (insert s1 x n) e1

eval n (ELift v1) = v1

eval n (EOp2 op e1 e2) =
        -- Evaluate e1 and e2 first
    let v1 = eval n e1
        v2 = eval n e2
        -- This case syntax allows us to select for the right op with the right
        -- value types at the same time.
    in case (op, v1, v2) of
        (Add, VFloat a, VFloat b) -> VFloat $ a +  b
        (Equ, VBool  a, VBool  b) -> VBool  $ a == b
        (Equ, VFloat a, VFloat b) -> VBool  $ a == b
        (Mul, VFloat a, VFloat b) -> VFloat $ a *  b
        (Neq, VBool  a, VBool  b) -> VBool  $ a /= b
        (Neq, VFloat a, VFloat b) -> VBool  $ a /= b
        _                         -> error "Type mismatch in eval/EOp2"

-- Resolving references means getting the value from the environment by name.
eval n (ERef s1) = n ! s1
```

Listing 32: ADT Evaluator