Not-Quite-Supersonic AD
Implementing forward and reverse mode automatic differentiation in the Futhark interpreter

Bachelor's thesis

39 pages

Marcus Jensen (pwt746)
Supervised by: Troels Henriksen

August 12th, 2024

Abstract

This thesis covers the theory behind- and the implementation of forward and reverse mode automatic differentiation in the Futhark interpreter. Throughout the text, the interpreter is incrementally improved until it matches that of the Futhark compiler. The implementation is then tested for correctness, and benchmarked, and its efficiency is evaluated. Finally, the thesis outlines the steps needed to be taken before the implementation can be merged into the Futhark code base.

Contents

1 Introduction

A crucial step in performing machine learning (ML) is calculating the gradient of a loss function. In order to do this, various partial derivatives of the loss function must be known. While it is entirely possible to write code which calculates these by hand, the complexity of modern ML models make this process quite slow and tedious. Instead, an approach called Automatic Differentiation (AD) is often used, as it can automatically calculate the derivatives of functions in computer programs. [1]

Futhark is a programming language which seeks to simplify the development of highly parallel code, such as that used to train ML models. [2] Naturally, the Futhark compiler supports AD, [5][6] however, the AD implementation for the Futhark interpreter has been sidelined for over two years.[1]

This report details the implementation of AD in the Futhark interpreter. Throughout the report, the implementation will be incrementally updated until it is capable of handling the same cases as the compiler. It will then be tested for correctness, and benchmarked. The goal of this project is to provide a clear path for AD to be implemented in the Futhark interpreter.

The final implementation can be found on GitHub.[2]

[1] `https://github.com/diku-dk/futhark/commit/d8a2b32d00c2889e0372da1822f9601dc69e99dc`
[2] `https://github.com/vox9/futhark/tree/interpreter-ad`

2 Background

2.1 Automatic differentiation

Automatic Differentiation (AD) [1] is a set of techniques, which compute the derivative of a function in a program. While several modes of AD exist, this project purely focuses on *forward mode*, and *reverse mode* derivation, as these are supported by the Futhark compiler. Both modes leverage the property that all scalar calculations in a program are a composition of a limited set of elementary mathematical operations, referred to as *primitive operations*; If the composition solely includes differentiable operations, its derivative can be computed using the multivariate chain rule, and the partial derivatives of its primitive operations. As these are well known, the problem becomes trivial.

Unlike *symbolic differentiation*[3], AD does not yield the derived function. Instead, it finds the value of the derivative for a given input by performing a nonstandard interpretation of the program, where each scalar is augmented with various information needed to find its derivative. Whenever such a value is passed to a primitive operation, its scalar half, now called its *primal value*, is calculated as it normally would be, while the other half is updated accordingly. Finally, the differentiated value can be computed.

Example 1

In order illustrate forward mode and reverse mode AD, consider deriving the function

```
f(x, y) = ((if x > 2 then x + 2 else -x), (reduce (*) [x, y, x]))
```

at `x = 3; y = 2`. Note that the function outputs a tuple of two scalars.

AD can be performed knowing only a trace of the primitive operations, which are applied during the evaluation of a function, as well as their inputs. This can be represented as a series of intermediary variables. The following is a trace of `f(3, 2)`:

e_1 = x (= 3)
e_2 = 2
e_3 = e_1 + e_2
e_4 = y (= 2)
e_5 = e_1 * e_4
e_6 = e_5 * e_1

Figure 1: The intermediary primitive operations, and their inputs when evaluating `f(3, 2)`. The expressions are given in the order of which they were evaluated, and the outputs of the function are e_3 and e_6

Deriving expressions of parameters and literals (such as e_1, e_2, and e_4) is simple - if it corresponds to the variable, which it is being derived with respect to, its derivative is `1`, and otherwise, it is `0`. On the other hand, deriving expressions of primitive operations (such as e_3, e_5, and e_6) necessitates the use of the multivariate chain rule.

In order to understand how the multivariate chain rule can be applied to find the derivative

[3] Transforming an expression into its differential using the rules of differentiation.

of an expression, consider deriving $\texttt{e}_5$. The application of the chain rule on $\texttt{e}_5$ yields the following equation:

$$\frac{\partial \texttt{e}_5}{\partial \texttt{x}} = \frac{\partial \texttt{e}_5}{\partial \texttt{e}_1}\frac{\partial \texttt{e}_1}{\partial \texttt{x}} + \frac{\partial \texttt{e}_5}{\partial \texttt{e}_4}\frac{\partial \texttt{e}_4}{\partial \texttt{x}}$$

As $\texttt{e}_5$ expresses the application of multiplication, and multiplication has two operands, applying the multivariate chain rule yields two terms. Each term is the product of a partial derivative of the primitive operation (multiplication) with respect one of its operands, and the partial derivative of the operand with respect to the, which is being derived with respect to. The former will be called the partial derivatives of the operation (in this case the partial derivatives of multiplication), and can be calculated using only the primal values of its operands and a set of well-known functions. The latter will be called partial derivatives with respect to a variable, and are either derivatives of parameters or literals, or expressions of primitive operations, in which case the chain rule must be applied to them.

Thus, deriving expressions of applications of primitive operators requires knowing the primal values, and derivatives of their operators. Deriving an expression therefore requires deriving, and often evaluating, all of its dependencies.

2.1.1 Forward mode

As the dependencies of a derivative mirror those of its expression, one strategy is to calculate each derivative alongside its expression. [1] This can be performed in any order, which satisfies the dependencies of the expression - including that, which is followed by the interpreter. Thus, augmenting each scalar with its derivative provides enough information to calculate the resulting derivative of the application of a primitive operation. This is how forward mode AD works.

The following is an example of how forward mode AD is performed to derive every expression in the trace of `f(3, 2)` (seen in figure 1) with respect to `x`:

$$\texttt{e}_1 \texttt{ = x = 3} \quad | \quad \frac{\partial \texttt{e}_1}{\partial \texttt{x}} \texttt{ = seed(x) = 1}$$

Forward mode AD starts by evaluating the first expression of the trace. While the derivative of this expression may seem trivial, it is actually not computed. Instead a variable is assigned its initial derivative. This way, when forward mode AD is run on multivariate functions, the partial derivative, which must be found, can be selected by assigning each variable a one-hot-encoded derivative. These are called the *seed values* of forward mode AD. As the derivative with respect to `x` is desired, the seed value of `x` is `1`.

The primal value, and the derivative of $\texttt{e}_1$ have now been computed, and the algorithm can move on to the next expression.

$$\texttt{e}_2 \texttt{ = 2} \quad | \quad \frac{\partial \texttt{e}_2}{\partial \texttt{x}} \texttt{ = 0}$$

This expression is a constant value. The derivative of a constant is `0`.

$$\texttt{e}_3 \texttt{ = } \texttt{e}_1 \texttt{ + } \texttt{e}_2 \texttt{ = 5} \quad | \quad \frac{\partial \texttt{e}_3}{\partial \texttt{x}} \texttt{ = } \frac{\partial \texttt{e}_3}{\partial \texttt{e}_2}\frac{\partial \texttt{e}_2}{\partial \texttt{x}} \texttt{ + } \frac{\partial \texttt{e}_3}{\partial \texttt{e}_1}\frac{\partial \texttt{e}_1}{\partial \texttt{x}}$$
$$\texttt{= 1 * 0 + 1 * 1}$$
$$\texttt{= 1}$$

As $\texttt{e}_3$ is an application of a primitive operation, the multivariate chain rule is used. It produces four derivatives, which must be solved: Two are the derivatives of the operands, and have thus already been found; and the remaining two are the partial derivatives of the primitive operation. These are well-known, and hardcoded into the AD implementation with some mapping to their respective operations. They can be looked up, and are always calculable using only the primitive values of the operands. The partial derivatives of `a + b` are `1`, and `1` respectively.

The derivatives of the chain rule can now be substituted by their values, and the derivative of the expression can be calculated.

$$\texttt{e}_4 \texttt{ = y = 2} \quad | \quad \frac{\partial \texttt{e}_4}{\partial \texttt{x}} \texttt{ = seed(y) = 0}$$

This is another expression of a variable, so its derivative can be found using the same method as for $\texttt{e}_1$. As the variable is `y` this time, and the derivative is being taken with respect to `x`, its seed value is `0`.

$$\texttt{e}_5 \texttt{ = e}_1 \texttt{ * e}_4 \texttt{ = 6} \quad | \quad \begin{aligned} \frac{\partial \texttt{e}_5}{\partial \texttt{x}} &= \frac{\partial \texttt{e}_5}{\partial \texttt{e}_1}\frac{\partial \texttt{e}_1}{\partial \texttt{x}} + \frac{\partial \texttt{e}_5}{\partial \texttt{e}_4}\frac{\partial \texttt{e}_4}{\partial \texttt{x}} \\ &= \texttt{e}_4 \texttt{ * 1 + e}_1 \texttt{ * 0} \\ &= \texttt{2} \end{aligned}$$

As the expression is of the application of a primitives operation, the same methodology can be used as for $\texttt{e}_3$. The respective partial derivatives of `a * b` are `b` and `a`.

$$\texttt{e}_6 \texttt{ = e}_5 \texttt{ * e}_1 \texttt{ = 18} \quad | \quad \begin{aligned} \frac{\partial \texttt{e}_5}{\partial \texttt{x}} &= \frac{\partial \texttt{e}_6}{\partial \texttt{e}_6}\frac{\partial \texttt{e}_1}{\partial \texttt{x}} + \frac{\partial \texttt{e}_6}{\partial \texttt{e}_1}\frac{\partial \texttt{e}_1}{\partial \texttt{x}} \\ &= \texttt{e}_1 \texttt{ * 2 + e}_5 \texttt{ * 1} \\ &= \texttt{12} \end{aligned}$$

This is the same case as $\texttt{e}_5$.

The partial derivative of of both outputs of `f(3, 2)` ($\texttt{e}_3$ and $\texttt{e}_6$) with respect to `x` have now been found. They are `1`, and `12` respectively. In order to find their partial derivatives with respect to `y`, the same method can be followed, setting the seed values of `x` and `y` equal to `0` and `1` respectively. This, however, requires calculating each of the derivatives again, as none of the partial derivatives with respect to `x` can be used to calculate those with respect to `y`.

The time complexity of forward mode AD can be calculated as follows: Let n be the total number of operations, and m the number of primitive operations, in a function, f, whereof the partial derivatives of p parameters must be found. As the number of primitive operations counts towards the total number of operations in f, $m \leq n$. In order to compute the primal values, and follow the control flow of f, the function must be run normally. This is achieved in $O(n)$ operations. Furthermore, each primitive operation is augmented to calculate its partial derivative with respect to each of the p parameters. The calculation of a partial derivative with respect to a variable is done using the chain rule, requiring the computation of every partial derivative of a primitive operation. Each can be computed in $O(1)$ operations, whereafter they are multiplied together with the partial derivative of

their respective operand, and summed together. The maximum number of operands for a primitive operation is a constant, $k \in O(1)$. Thus, computing one partial derivative with respect to a variable takes $O(1) \cdot O(k) + 2 \cdot O(k) \in O(1)$ operations. Computing the partial derivatives with respect to p variables takes $p \cdot O(1) \in O(p)$ operations. As it must be done for each of the m primitive operations, it takes $m \cdot p \cdot O(1) \in O(m\ p)$ operations. Thus, the time complexity of forward mode AD is $O(n + m\ p) \in O(n\ p)$.

2.1.2 Reverse mode

Whereas forward mode tackles derivation in the same order as the expression would be evaluated, it is also possible to calculate the derivatives of a function starting at its outputs. [1] Here, the values of the partial derivatives of primitive operations are calculated using the primal values of their operands, while partial derivatives with respect to variables are expanded using the chain rule. This continues until an expression with no dependencies is reached, such as one of a parameter, or a literal. Reverse mode AD method poses the challenge that the primal values, as well as the primitive operations used to compute them, are accessed in the opposite order of their evaluation. They must therefore be saved when the function is evaluated. This is the data used to augment each scalar when performing reverse mode AD. The reward for the extra work is the ability to calculate every partial derivative of a single output in $O(n)$ time.

While there are many possible representations of the augmented data, this report will use that of a computation graph, as it is simple to explain. A computation graph can be constructed from a trace of primitive operations by going through each line of the trace in the order, they are evaluated, and creating a node for its respective intermediate value. If a node represents a variable, its name must be saved; and if it represents an application of a primitive operation, the operation must be saved, and directed edges drawn connecting it to the nodes representing each of its operands. The edges must be enumerated by the order of which their respective operand appears in the expression, as it determines which partial derivative of the operation, their derivatives are multiplied with when applying the chain rule.

A precise algorithm for constructing a computation graph can be found in section 3.2.

The computation graph for the trace of `f(3, 2)` (seen in figure 1) is:

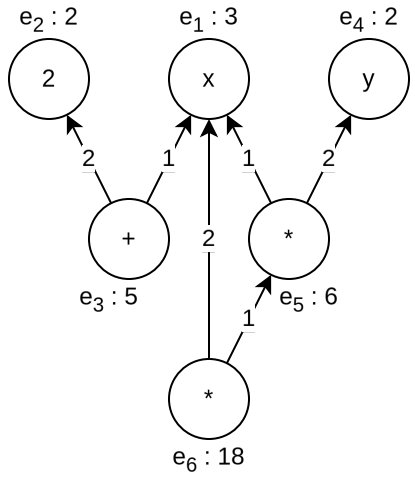


Figure 2: The computation graph of the trace of `f(3, 2)`.

The partial derivatives of a given output can now be computed by propagating a rolling product of the partial derivatives of each primitive operation through the graph. This value is called called a *sensitivity*. Its initial value, `1`, is the seed value in reverse mode AD. In order to illustrate how the sensitivity is propagated through the computation graph to calculate the partial derivatives of $\texttt{e}_6$, consider the following visualization:

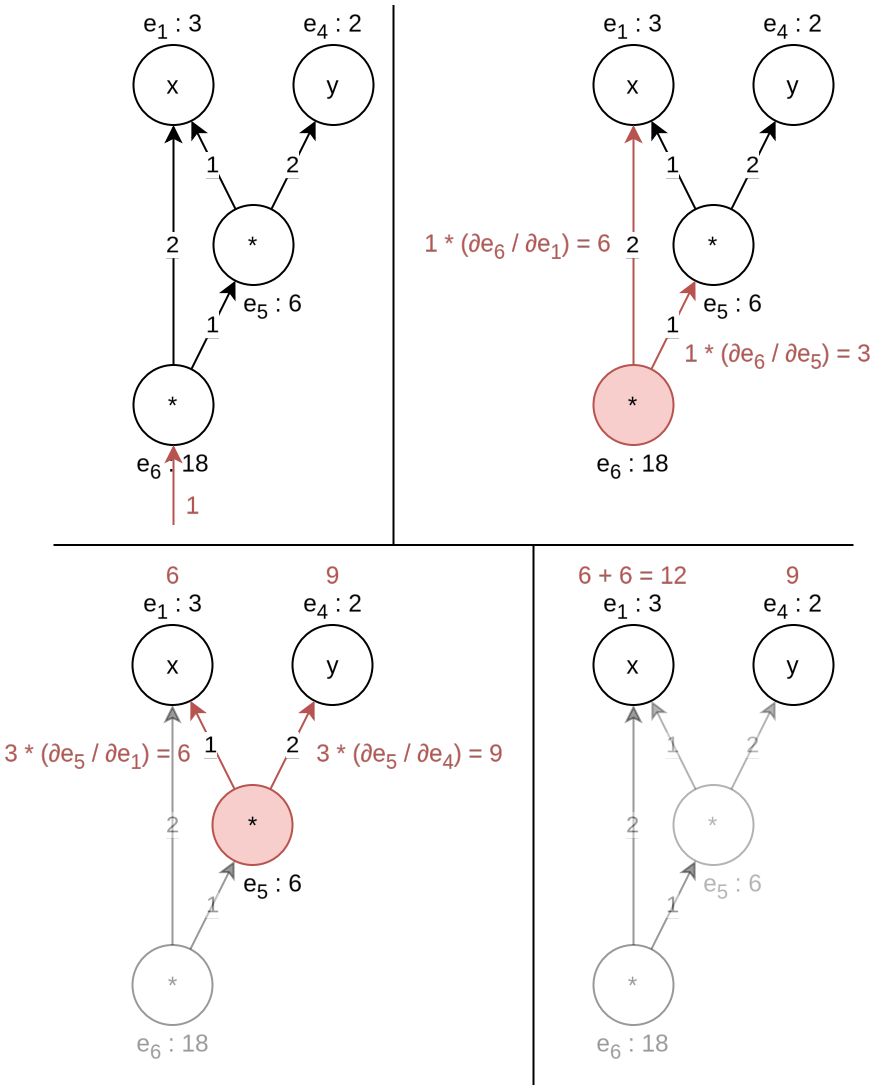


Figure 3: Computing every partial derivative of e_6 in the trace of `f(3, 2)` using reverse mode AD.

First, the output, which is being derived, (e_6) is assigned its seed value, 1 (top left). Then, the values of the partial derivatives of its operator are multiplied with the seed value. This produces one sensitivity for each operand, which is passed along to its respective node in the graph (top right). When a sensitivity reaches a parameter, its value is saved, as it has now been propagated through the entire graph. If, instead, it reaches another primitive operation, it is multiplied by each of the partial derivatives of the new operation, and the resulting sensitivities are, again, propagated to their respective operands (bottom left). When a sensitivity reaches the node of a parameter, which already has a sensitivity assigned to it, the sum of the two are taken. This is because each path essentially corresponds to one term in the chain rule. Reverse mode AD is finished when the partial derivatives of every

primitive operation have been calculated, and passed along (bottom right).

The partial derivatives of $\mathtt{e}_6$ can now be extracted by reading the sums of the sensitivities, which are saved in the nodes of the parameters. The derivative of $\mathtt{e}_6$ with respect to `x` is 12, while the derivative of $\mathtt{e}_6$ with respect to `y` is 9.

The time complexity of reverse mode AD can be calculated like so: Let n be the total number of operations, and m the number of primitive operations, in a function, f, whereof the partial derivatives of o outputs must be found. As the number of primitive operations counts towards the total number of operations in f, $m \leq n$. In order to compute the primal values, and follow the control flow of f, the function must be run normally. This is achieved in $O(n)$ operations. Furthermore, each primitive operation is augmented to update the computation graph, which is returned along with its output. This can be done in $O(1)$ time. As it must be done for each of the m primitive operations, the first run of the function takes $O(n) + O(m) \cdot O(1) \in O(n + m)$ time. The sensitivity must now be propagated through the computation graph of each of the outputs. The computation graph of an output can contain at most as many primitive operations as the function it was generated from. It can therefore contain no more than m primitive operations. For each primitive operation, its respective partial derivatives must be calculated and multiplied with a sensitivity. The maximum number of operands for a primitive operation is a constant, $k \in O(1)$, and each partial derivative can be calculated and multiplied with a sensitivity in $O(1)$ time. Thus, the computation of the partial derivatives of the operations, and their multiplication with the sensitivities takes $O(k) \cdot O(1) + O(k) \in O(1)$ operations. Furthermore, each primitive operation will at most take one parameter per operator, so at most k numbers will have to be saved or updated per. primitive operation, requiring a maximum of $O(m\ k) \in O(m)$ operations per output. Thus, calculating the partial derivatives of o outputs takes $O(n + m + m\ o) \in O(n\ o)$ operations.

—**End of example**———————————————————————————————

2.2 Futhark

Futhark [2] is a purely functional programming language, created to simplify the development of highly parallel numerical software. Its design hinges on keeping the language small, such that numerous aggressive optimizations can be applied to its code. Thus, the resulting programs turn out fairly well-optimized without requiring the programmer to consider hardware-specific optimizations.

While Futhark is primarily a source-to-source compiler targeting various languages that run on the CPU or GPU, it also implements an interpreter. Both are written in Haskell, and have a shared code base. Whereas the compiler produces well-optimized code, the intended use of the interpreter is testing and debugging code, making its speed less important.

As this project purely focuses on the Futhark interpreter, the following sections will describe its general structure.

2.2.1 Representing values

Futhark has built-in support for floats, integers, and booleans, as well as arrays, records, and tuples. In order to operate on any of these values, the interpreter must represent them

using a mapping to their Haskell-equivalents. This is achieved using a type, `Value`, with the following constructors: [3]

- `ValuePrim`, which represents a *primitive value* - a non-composite value, which is built into the language,
- `ValueArray`, which represents an array,
- `ValueRecord`, which represents a record, or a tuple,
- `ValueFun`, which represents a function and,
- `ValueSum`, which represents a sum type.

2.2.2 Evaluating a program

The evaluation model of the Futhark interpreter is fairly conventional for a statically scoped, purely functional language. [3] After lexing, parsing, and type annotation, a program is interpreted by traversing its abstract syntax tree. Every function of the program is given a single expression for its output value. Running a program entails evaluating the expression of the entry point, and returning the result. Futhark uses call-by-value semantics, and is free of side effects, aside from non-termination.

A Futhark program consists of a sequence of declarations, which are placed in the environment of the interpreter. The environment also contains a set of built-in declarations, many of which are implemented in Haskell, acting as basic building blocks which enable otherwise impossible functionality in Futhark. Those of them, which return `ValuePrim`s can be considered the primitive operations of Futhark. Note that second order functions do not fall under this definition, as they don not strictly return `ValuePrim`s. Instead, their output type match that of the function, they are given.

2.2.3 Automatic differentiation in Futhark

In Futhark, AD is initiated using one of two functions - `jvp`, which performs forward mode AD, and `vjp`, which performs reverse mode AD. [5][6] Both take the function to be derived, along with its input values as parameters. Furthermore, they are given the seed values of their respective mode, which act as initial values for the derivatives when AD is performed. In practice, these should be given as a set of one-hot encoded values, selecting which partial derivative to return, in case of `jvp`, or which output to derive, in case of `vjp`.

`jvp` returns the same output as its given function, with each scalar value replaced by its derivative. `vjp` returns the same input as its given function, with every scalar value replaced by its derivative.

Example 2

To understand how `jvp` and `vjp` are used to perform AD in Futhark, examine the following cases where two functions, $f(x, y)$, and $g(x)$, are derived:

To find the derivative of $f(x, y)$ with respect to x, `jvp` can be used as follows:

```
jvp f (x, y) (1, 0)
```

The first parameter, `f`, is the function, while the second parameter, `(x, y)`, is a tuple containing the parameters. The final parameter is a tuple of the seeds for each parameter. Since the derivative of $f(x, y)$ with respect to x is desired, the seed of `x` is `1`, and the seed of `y` is `0`.

If, instead, both the derivatives of $f(x, y)$ and $g(x)$ are desired with respect to y, they can be found with a single run to `jvp`:

```
jvp (\(x, y) -> (f(x, y), g(x))) (x, y) (0, 1)
```

As `jvp` returns the derivative of every output with respect to a single variable, `jvp` is run on a lambda function, which returns a tuple of $f(x, y)$ and $g(x)$. As x and y are still the only variables present, the second parameter remains unchanged. The seeds change, as the derivative with respect to y is now desired. The seed of `x` is now `0`, and the seed of `y` is `1`.

Finally, if both the partial derivative of $f(x, y)$ with respect to x and y are desired, `vjp` must be used like so:

```
vjp f (x, y) 1
```

The first parameter is the function, `f`, and the second parameter is the inputs of the function, `(x, y)`. The final parameter is the seeds of the outputs of the function. Since $f(x, y)$ outputs a single scalar value, there is only one seed, and since the partial derivatives of x and y with respect to the output of `f` are desired, it must be `1`.

—**End of example**———————————————————————————————

3 Implementation

The following part describes the implementation of forward and reverse mode AD. As the full Futhark implementation includes many details, which are not relevant to portraying the core idea of the algorithm, the implementation will be presented for a simplified language with the following properties:

- Like Futhark, the language is purely functional with no side effects.
- All values in the language are untyped scalars.
- AD must be implemented for an interpreter of the language. The interpreter delegates certain tasks to various functions, including:
  - `doOp(op, p1...pn)`, which is used to apply a primitive operation (`op`) to a list of parameters ($p_1 \ldots p_n$). Before the interpreter is extended to support AD, the implementation of `doOp` is simply `return op(p1...pn)`.
  - `evalJvp` and `evalVjp`, which are called to initiate forward mode, and reverse mode AD respectively. They are initially undefined.
  - `getPartials(op)`, which returns a list of the partial derivatives of a primitive operation (`op`) with respect to each of its operands.
  - `eval(exp)`, which evaluates an expression (`exp`) using the interpreter, and returns the result.

3.1 Forward mode

Forward mode AD can be boiled down to three steps:

1. Augmenting each scalar parameter of the input function with its seed value.
2. Performing a non-standard interpretation of the input function, where the application of primitive operations is augmented to calculate the derivatives of their outputs.
3. Returning the derivative of each scalar output.

Steps 1 and 3 In order to augment scalars, the value type of the interpreter must be turned into a sum type. Instead of being only scalars, values can now either hold a `Primitive(v)`, which consists of a single scalar value (`v`), or a `JvpValue(p, d)`, which consists of two scalars - a primal value (`p`), and its derivative (`d`). In order to incorporate this change, all code in the interpreter must be updated to deal only in `Primitive`s.

The interpreter now uses `Primitive` values, and the program should run just as it did before. If, however, a `JvpValue` is introduced, the language will not be able to use it, as all functions are only defined for `Primitive`s. The primal value of a `JvpValue` must be treated just the same as a `Primitive`, so this must be resolved.

All functions which take `Primitive`s as parameters can be categorized into one of two groups: Those which are derivable, and those which are not. For the sake of simplicity, all primitive operations are assumed to be derivable, while all other built-in functions are not. The latter can be made to treat `Primitive`s and `JvpValue`s the same by introducing a function, `getValue(v)`, which, given a `Primitive`, returns its value, and given a `JvpValue`,

returns its primal value. Apart from running all scalar inputs through this function, the non-derivable functions will run as they did before, and any scalar values they output will be treated as constants. This matches the behavior of the Futhark compiler. (An alternative approach could be to throw an error when a `JvpValue` was passed to a non-derivable function, however this would make the language more restrictive. Furthermore, the output of a non-derivable function may never be returned, but only used intermediately, for example in control flow.) Meanwhile, derivable functions (and thus primitive operations) must return a `JvpValue` if their return value has a derivative which is non-zero, so as not to lose information. The augmentation of primitive operations will be handled later.

A `Primitive` can now be augmented into a `JvpValue` using the following function:

```
augmentJvp(Primitive(v), d):
1: return JvpValue(v, d)
```

The function takes a `Primitive` value, and extracts its scalar (`v`). Furthermore, it takes the seed value of `v` (`d`) - its initial derivative, declared by the programmer. Note that the function is only defined for `Primitive`s. It will be extended to handle all values in section 3.3.

To extract the derivative of a value, another function is defined.

```
deriveJvp(JvpValue(p, d)):
1: return d
deriveJvp(Primitive(v)):
1: return 0
```

Given a `JvpValue`, it simply returns its derivative (`d`). It is, however, also possible that a function returns a `Primitive` value when performing forward mode AD. As all inputs are augmented into `JvpValue`s, and all primitive operations will later be modified to return a `JvpValue` if any of their parameters was a `JvpValue`, this only happens when the output of the function is not the result of a computation involving `JvpValue`s. In this case, the derivative of the output is 0, and thus, the derivative of a `Primitive` value is 0.

The `evalJvp` function can now be defined.

```
evalJvp(f, v_1...v_n, s_1...s_n):
    ▷ Step 1: Augment each input value with its respective seed value
1: v'_1...v'_n ← map(augmentJvp, zip(v_1...v_n, s_1...s_n))
    ▷ Step 2: Evaluate the function (through the interpreter) on the augmented inputs
2: o_1...o_m ← eval(f(v'_1...v'_n))
    ▷ Step 3: Extract the derivative of each output
3: return map(Primitive ∘ deriveJvp, o_1...o_m)
```

As `augmentVjp` only accepts `Primitive`s, $v_1 \ldots v_n$ must all be `Primitive` values. `evalJvp` will later be extended to handle all values.

Step 2 As all primitive operations are handled by `doOp`, they can be augmented by modifying the function like so:

```
doOp(op, p_1...p_n):
1: pv_1...pv_n ← map(getValue, p_1...p_n)
2: pv ← op(pv_1...pv_n)
```

3: **if every value in** $\texttt{p}_1 \ldots \texttt{p}_n$ **is a** `Primitive` **then**
4: | **return** `Primitive(pv)`
5: **else**
6: | $\texttt{pd}_1 \ldots \texttt{pd}_n \leftarrow$ `getPartials(op)`
7: | $\texttt{dv}_1 \ldots \texttt{dv}_n \leftarrow$ `map(deriveJvp,` $\texttt{p}_1 \ldots \texttt{p}_n$`)`
8: | $\texttt{dv} \leftarrow \texttt{dv}_1 \cdot \texttt{pd}_1(\texttt{pv}_1 \ldots \texttt{pv}_n) + \ldots + \texttt{dv}_n \cdot \texttt{pd}_n(\texttt{pv}_1 \ldots \texttt{pv}_n)$
9: | **return** `JvpValue(pv, dv)`

The function starts by extracting the primal value of every parameter ($\texttt{pv}_1 \ldots \texttt{pv}_n$) using `getValue` (line 1). It then applies the primitive operation (`op`) to the primal values (line 2), yielding the primal value of the expression (`pv`). The function now splits into two branches. The first branch (line 4) is only run, if every parameter is a `Primitive`. In this case, the derivative is always 0, as the expression does not contain the value, it is being derived with respect to. Thus, the primal value can be returned as a `Primitive`. If AD is never initiated, this is the only branch, which will be reached by the interpreter.

The second branch (lines 6-9) runs if one or more parameters is a `JvpValue`. In this case, the partial derivatives of the primitive operation ($\texttt{pd}_1 \ldots \texttt{pd}_n$) are found using `getPartials` (line 6). Furthermore, the derivative of every parameter ($\texttt{dv}_1 \ldots \texttt{dv}_n$) is extracted using `deriveJvp` (line 7). The derivative (`dv`) can now be computed using the multivariate chain rule, as well as the partial derivatives (line 8). Finally, a `JvpValue` containing the primal value, and the derivative is returned (line 10).

—Example 3

To understand exactly how this implementation performs forward AD, consider the following example, where the interpreter must compute the derivative of $y \cdot x \cdot x + (2+2)$ with respect to x, and $x = 3$ and $y = 2$.

The interpreter would evaluate the expression:

```
jvp (λ(x, y) → y * x * x + (2 + 2)) (3, 2) (1, 0)
```

In step 1 of `evalJvp`**,** the input variables (`[Primitive(3), Primitive(2)]`), and the seeds (`[Primitive(1), Primitive(0)]`) are be paired up, and augmented using `augmentJvp`. This produces the values `[JvpValue(3, 1), JvpValue(2, 0)]`.

In step 2 of `evalJvp`**,** the function (`f = λ(x, y) → y * x * x + (2 + 2)`) is run through the interpreter, passing the augmented values to their respective parameters. The interpreter must now evaluate the following expression:

```
JvpValue(2, 0)*JvpValue(3, 1)*JvpValue(3, 1) + (Primitive(2)+Primitive(2))
```

The primitive operations can, of course, be applied in any order, which satisfies the order of operations. The following is one possible trace of every execution of `doOp` (and thus every application of a primitive operation), when evaluating the expression. Note that the output of an operation is denoted by the name of the operation surrounded by square brackets.

	Operation 1	Operation 2	Operation 3	Operation 4
Expression	`JvpValue(2, 0) * JvpValue(3, 1)`	`[Operation 1] * JvpValue(3, 1)`	`Primitive(2) + Primitive(2)`	`[Operation 2] + [Operation 3]`
Operation	`*`	`*`	`+`	`+`
Branch	Second	Second	First	Second
$\texttt{pv}_1 \ldots \texttt{pv}_n$	`[2, 3]`	`[6, 3]`	`[2, 2]`	`[18, 4]`
$\texttt{dv}_1 \ldots \texttt{dv}_n$	`[0, 1]`	`[2, 1]`	N/A	`[12, 0]`
$\texttt{pd}_1 \ldots \texttt{pd}_n$	`[λ(a, b) → b, λ(a, b) → a]`	`[λ(a, b) → b, λ(a, b) → a]`	N/A	`[λ(a, b) → 1, λ(a, b) → 1]`
`pv`	`2 * 3 = 6`	`6 * 3 = 18`	`2 + 2 = 4`	`18 + 4 = 22`
`dv`	`0 * 3 + 1 * 2 = 2`	`2 * 3 + 1 * 6 = 12`	N/A	`12 * 1 + 0 * 1 = 12`
Output	`JvpValue(6, 2)`	`JvpValue(18, 12)`	`Primitive(4)`	`JvpValue(22, 12)`

Thus, the output of the function is `JvpValue(22, 12)`.

In step 3 of `evalJvp`, the output is run through `deriveJvp`, yielding the value 12. The value is represented by a `Primitive`, making the output of `evalJvp` `Primitive(12)`. This matches the value of the derivative.

—**End of example**———————————————————————————————

3.2 Reverse mode

The basic steps of performing reverse mode AD are:

1. Augmenting each scalar parameter of the input function with an initial representation of its computation graph.
2. Performing a non-standard interpretation of the input function, where the application of primitive operations is augmented to update combine the computation graphs of each operand.
3. Calculating every partial derivative of one or more outputs by propagating through their computation graphs in reverse topological order.

Step 1 As with forward mode AD, the value type of the interpreter must be extended with a new constructor - `VjpValue`. Before this can be done, a new type must be created to store the computation graph. This type will be called `Tape`, and needs only three constructors.

- `TapeVar(id, v)` represents a parameter of of input function. These act as free variables when performing reverse mode AD. In order to tell variables apart, each is given a unique `id`. Furthermore, their value, `v`, is stored.
- `TapeConst(v)` represents a constant value, `v`.
- `TapeOp(op,` $\texttt{tp}_1 \ldots \texttt{tp}_n$`, v)` represents the application of a primitive operation, `op`, on the values $\texttt{tp}_1 \ldots \texttt{tp}_n$ (represented as tapes). The primal value of the result is stored in `v`.

As every `Tape` stores a primal value, `VjpValue` need not store a primal value itself, but only a `Tape`. A function named `getTapeValue(tp)` can be implemented, performing pattern matching on a `Tape` (`tp`) and returning its primal value, `v`. This can, in turn, be used to implement `getValue(v)` for `VjpValue`s.

`Primitive`s can now be augmented into `VjpValue`s with the following function:

```
augmentVjp(Primitive(v), id):
 1: return VjpValue(TapeVar(id, v))
```

It simply extracts the value of the `Primitive` (`v`), and wraps it in a `TapeVar` along with an `id`. This creates the `Tape` representation of a variable. It is then wrapped in a `VjpValue` and returned.

The first two steps of `evalVjp` can now be defined as:

```
evalVjp(f, v_1...v_n, s_1...s_m):
    ▷ Step 1: Augment each input value
 1: v'_1...v'_n ← map(augmentVjp, enumerate(v_1...v_n))
    ▷ Step 2: Evaluate the function (through the interpreter) on the augmented inputs
 2: o_1...o_m ← eval(f(v'_1...v'_n))
    ▷ (...)
```

Its implementation closely resembles that of `evalJvp`, however instead of passing seed values to `augmentVjp`, the parameters are enumerated. This assigns each a unique ID.

Step 2 As with forward mode AD, the augmentation of primitive operations is implemented by modifying `doOp`. For simplicity, reverse mode will be implemented without considering the implementation of forward mode. The two modes will be brought together in section 3.3.

```
doOp(op, p_1...p_n):
 1: pv_1...pv_n ← map(getValue, p_1...p_n)
 2: pv ← op(pv_1...pv_n)
 3: if every value in p_1...p_n is a Primitive then
 4:     return Primitive(pv)
 5: else
 6:     tp_1...tp_n ← map(λp → match p
                              with Primitive(v):  TapeConst(v)
                              with VjpValue(tp):  tp
                            end, p_1...p_n)
 7:     return VjpValue(TapeOp(op, tp_1...tp_n, pv))
```

Like its forward mode-enabled counterpart, the function starts by extracting the primal value of every operator ($\texttt{pv}_1 \ldots \texttt{pv}_n$), and applying the primitive operation to them to calculate the primal value of the expression (`pv`) (lines 1 and 2). If every parameter is a `Primitive`, it returns a `Primitive` without storing the computation graph (line 4). Essentially, this acts as constant propagation. It ensures that the computation graph will never contain a primitive operation applied only to constants, as the resulting derivative always would be 0. Therefore, the operation may as well be computed, and treated as a constant, which may later be added to the computation graph.

The second branch (lines 6 and 7) runs if one or more parameters is a `VjpValue`. In this

case, every parameter is turned into its `Tape` representation (lines 6). `Primitive` values are turned into `TapeConst`s, while `VjpValue`s are unwrapped, extracting their tape. Finally, a `TapeOp` is returned, storing the primitive operation, which was applied (`op`), the tapes of the parameters it was applied to ($\texttt{tp}_1 \ldots \texttt{tp}_n$), and the primal value of the result (`pv`) (line 7). Note that, much like trees, wrapping a `Tape` in another `Tape` is assumed to be done using references, and not by copying, as this would make the operation much slower. One way to do this is by keeping `Tape`s in the heap, and only storing references to them in the `VjpValue` type. Haskell automatically performs such optimizations using persistent datastructures with shared structures.

Step 3 Each output of a function, which is given `VjpValue`s as parameters, is now either a `Primitive`, or a `VjpValue` containing a computation graph annotated with the primal value of each variable, constant, and application of a primitive operation. In the former case, every partial derivative of the output is 0, and in the latter, the resulting `Tape` must be derived. The derivation of a `Tape` can be implemented as follows:

```
deriveTape(tp, s, dv_1...dv_m):
 1: match tp
 2:     with TapeVar(id, v):
 3:         dv_id ← dv_id + s
 4:         return dv_1...dv_m
 5:     with TapeConst(v): return dv_1...dv_m
 6:     with TapeOp(op, tp_1...tp_n, v):
 7:         pv_1...pv_n ← map(λtp → getTapeValue(tp), tp_1...tp_n)
 8:         pd_1...pd_n ← getPartials(op)
 9:         pdv_1...pdv_n ← map(λpd → s · pd(pv_1...pv_n), pd_1...pd_n)
10:         dv_1...dv_m ← reduce(λ(dv_1...dv_m, tp, pdv) →
                                    deriveTape(tp, pdv, dv_1...dv_m)
                              , dv_1...dv_m
                              , zip(tp_1...tp_n, pdv_1...pdv_n))
11:         return dv_1...dv_m
```

The function is given three parameters - the tape to be derived (`tp`), the current rolling multiple of the partial derivatives of the primitive operations (`s`, i.e. the sensitivity), and a list of the values of the partial derivatives of `tp` with respect to each variable, indexed by the ID of the variable ($\texttt{dv}_1 \ldots \texttt{dv}_m$). This list is initially populated by zeros, and is modified throughout the function, before being returned with the proper values of the partial derivatives.

`deriveTape` consists of a match case deriving each type of `Tape` differently. The first case of the function captures `TapeVar`s. These represent free variables. In this case, the sensitivity has made its way to the top of the computation graph, and must be added to the value of the partial derivative with respect to the variable represented by the `TapeVar` - i.e. $\texttt{dv\_}id$. Finally, the now-modified $\texttt{dv}_1 \ldots \texttt{dv}_m$ is returned (line 4).

The second match case captures `TapeConst`s. As these represent constants, they do not modify the value of any partial derivative. Therefore, $\texttt{dv}_1 \ldots \texttt{dv}_m$ is returned unmodified (line 5).

The final match case captures `TapeOp`s - the applications of primitive operations. This is where the multivariate chain rule is applied. The first step in doing so is extracting the primal value of each of the operands (line 7). Then, the partial derivatives of the primitive operation are looked up (line 8). The values of the partial derivatives of the primitive operation can now be calculated using the primal values of the operands, and multiplied with the sensitivity (line 9). This calculates the new sensitivities ($\texttt{pdv}_1 \ldots \texttt{pdv}_n$) - one for each of the operands. The tapes of the operands ($\texttt{tp}_1 \ldots \texttt{tp}_n$) are then joined with their sensitivities ($\texttt{pdv}_1 \ldots \texttt{pdv}_n$) and passed along to a recursive call to `deriveTape`. This is done inside of a `reduce` statement, with $\texttt{dv}_1 \ldots \texttt{dv}_m$ being the initial value. Thus, each call to `deriveTape` can change the partial derivatives for each parameter, which is further along in the computation graph. Finally, the newly calculated partial derivatives are returned (line 11).

`deriveVjp` can now be defined as follows:

```
deriveVjp(VjpValue(tp), s, n):
1: if s ≠ 0 then
2:     return deriveTape(tp, s, repeat(0, n))
3: else
4:     return repeat(0, n)
deriveVjp(Primitive(v), s, n):
1: return repeat(0, n)
```

It takes the output value to derive, as well as its seed value (`s`), and the number parameters of the function (`n`). If the derived value is a `VjpValue` and the seed value is non-zero, it calls `deriveTape` on the tape (`tp`), the seed, and a list of `n` `0`s, which act as placeholders for the partial derivatives (line 2). Otherwise, every partial derivative is guaranteed to be `0`, and the function returns a list of `n` `0`s

Finally, the definition of `evalVjp` can be updated as follows:

```
evalVjp(f, v_1...v_n, s_1...s_m):
    ▷ Step 1: Augment each input value
1: v'_1...v'_n ← map(augmentVjp, enumerate(v_1...v_n))
    ▷ Step 2: Evaluate the function (through the interpreter) on the augmented inputs
2: o_1...o_m ← eval(f(v'_1...v'_n))
    ▷ Step 3: Derive the outputs of the function
3: return map(Primitive, join((+), map(λ(o, s) → deriveVjp(o, s, n)
                                        , zip(o_1...o_m, s_1...s_m))))
```

In step 3, every return value is joined together with its respective seed, and derived. This results in one list of partial derivatives per. output. The lists are then joined into one by adding the elements, which are on the same index (line 3). This is the way, the Futhark compiler handles reverse mode derivation, and thus, it was replicated for compatibility. If the seeds are one-hot encoded, this strategy results in a list of partial derivatives of the output, whose seed is `1`, with respect to every parameter.

—Example 4

In order to understand exactly how this implementation performs reverse mode AD, consider the following example, where the interpreter must compute every partial derivative of $y \cdot x \cdot x + (2 + 2)$ where $x = 3$ and $y = 2$.

The interpreter would evaluate the expression:

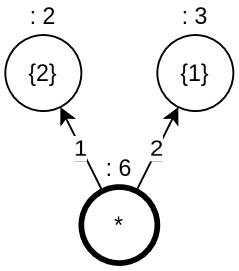
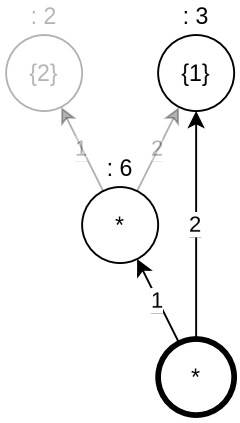
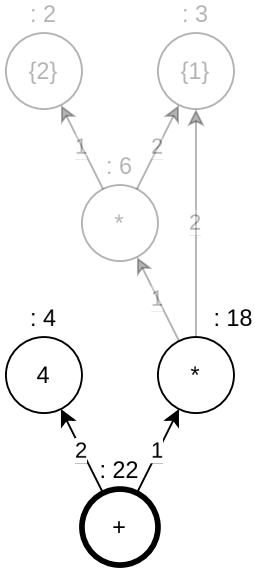
```
vjp ((x, y) → y * x * x + (2 + 2)) (3, 2) 1
```

In step 1 of `evalVjp`**,** the input variables (`[Primitive(3), Primitive(2)]`) are augmented using `augmentVjp`. This produces the values `[VjpValue(TapeVar(1, 3)), VjpValue(TapeVar(2, 2))]`.

In step 2 of `evalVjp`**,** the function (`f = (x, y) → y * x * x + (2 + 2)`) is run through the interpreter, passing the augmented values to their respective parameters. The interpreter must now evaluate the expression:

```
VjpValue(TapeVar(2, 2)) * VjpValue(TapeVar(1, 3)) * VjpValue(TapeVar(1, 3))
                        + (Primitive(2) + Primitive(2))
```

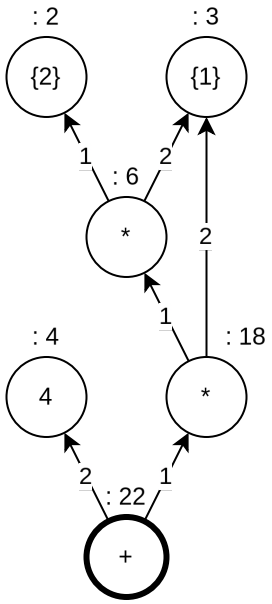
The following is one possible trace of every application of `doOp` (and thus every application of a primitive operation), when evaluating the expression. Note that the output of an operation is denoted by the name of the operation surrounded by square brackets. Furthermore, for brevity, all `VjpValue`s will be denoted only by their internal `Tape`.

	Operation 1	**Operation 2**	**Operation 3**	**Operation 4**
Expression	`TapeVar(2, 2) * TapeVar(1, 3)`	`[Operation 1] * TapeVar(1, 3)`	`Primitive(2) + Primitive(2)`	`[Operation 2] + TapeConst(4)`
Operation	`*`	`*`	`+`	`+`
Branch	Second	Second	First	Second
$\texttt{pv}_1 \ldots \texttt{pv}_n$	`[2, 3]`	`[6, 3]`	`[2, 2]`	`[18, 4]`
$\texttt{tp}_1 \ldots \texttt{tp}_n$	`[TapeVar(2, 2), TapeVar(1, 3)]`	`[[Operation 1], TapeVar(1, 3)]`	N/A	`[[Operation 1], TapeConst(4)]`
Output	`TapeOp(*, [TapeVar(2, 2), TapeVar(1, 3)], 6)`	`TapeOp(*, [[Operation 1], TapeVar(1, 3)], 18)`	`Primitive(4)`	`TapeOp(+, [[Operation 2], TapeConst(4)], 22)`
Output computation graph			N/A	

Thus, the output of the function is:

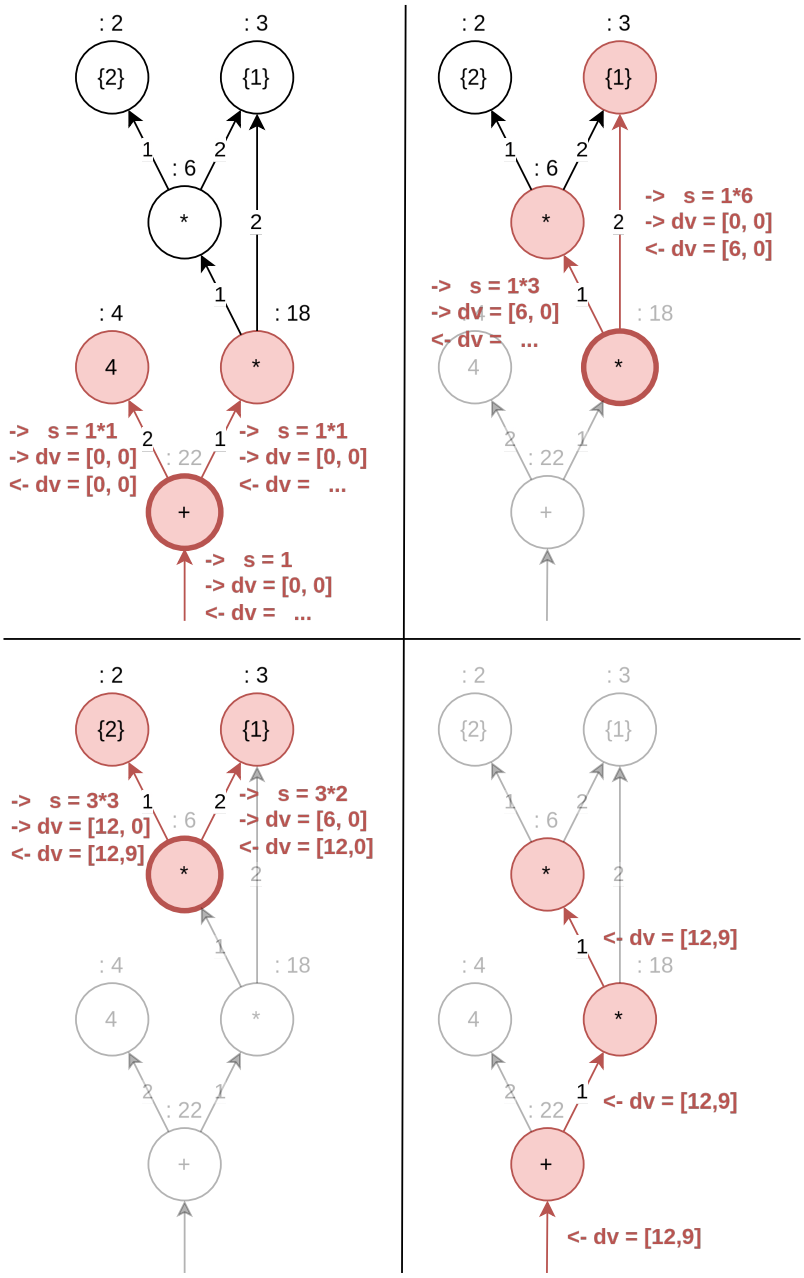
```
TapeOp(+, [
    TapeOp(*, [
        TapeOp(*, [
            TapeVar(2, 2),
            TapeVar(1, 3)
        ], 6),
        TapeVar(1, 3)
    ], 18),
    TapeConst(4)
], 22)
```

which corresponds to the following computation graph:



where the fat node is the output of the function, and each variable is marked by its ID wrapped in curly brackets. Note that the nodes are not named - the only information needed is the primal value of each node, and what variable or operation it represents, as well as which operator each of its connected nodes corresponds to.

In step 3 of `evalVjp`**,** the output is run through `deriveVjp` with the seed value `1`. As the seed value is nonzero, and the ouput is a `VjpValue`, its tape is passed on to `deriveTape`.



Derivation of the output starts at the fat node, where the sensitivity (`s`) is given its seed value, `1`, and the derivatives ($\texttt{dv}_1 \ldots \texttt{dv}_m$ aka `dv`) are initiated with the values `[0, 0]`. The fat node represents the application of the primitive operation `+`. Therefore, the chain rule is applied. The first step is to find the primal values of the operands of the operation (`pvn`). These are `[18, 4]`. Now, the partial derivatives of the operation ($\texttt{pd}_1 \ldots \texttt{pd}_n$) are looked up, and their values are calculated and multiplied with the sensitivity to obtain the new sensitivities. The partial derivatives of `+` are `1` and `1`, and the sensitivity is `1`. Thus, the

new sensitivities are `[1·1=1, 1·1=1]`. The sensitivities are now passed to `doOp` along with the tape of their respective operands, and the derivatives. This can be done in any order, but for illustrative purposes, it is easier to do starting at operand #2 (notice the numbers on the edges). Thus, `doOp` is called with the tape of operand #2, a sensitivity of `1`, and the derivatives `[0, 0]`. As operand #2 is a constant value, all this call does is return the derivatives unchanged. `doOp` can now be called on operand #1 with a sensitivity of `1` and the derivatives, which are still `[0, 0]` (top left).

Operand #1 is another application of a primitive operation - this time `*`. The primal values of the new operands are `[6, 3]`, and the partial derivatives of `a * b` are `b` and `a`. Thus, the new sensitivities are `[1·3=3, 1·6=6]`. Reduce is now called, calling `deriveTape` on (new) operand #2 with a sensitivity of `6`, and the derivatives `[0, 0]`. As operand #2 represents the parameter whose ID is `1`, it adds the sensitivity to the first number in the list of derivatives. Thus, it returns the new derivatives `[0+6=6, 0]`. `deriveTape` is now called on operand #1 with a sensitivity of `3`, and the derivatives, `[6, 0]`. (top right)

Operand #1 is also an application of multiplication. The primal values of its operands are `[2, 3]`. Thus, the new sensitivities are `[3·3=9, 3·2=6]`. `deriveTape` is now called on (new new) operand #2 with its sensitivity, `6`, and the derivatives, `[6, 0]`. As operand #2 also represents the parameter whose ID is `1`, it adds the sensitivity to index `1` of the list, returning the derivatives `[6+6=12, 0]`. `deriveTape` is now called on operator #1 with its sensitivity, `9`, and the updated derivatives, `[12, 0]`. As operator #1 represents the parameter whose ID is `2`, it adds the sensitivity to the second index of the list of derivatives, returning `[12, 0+9=9]` (bottom left).

The call to `deriveTape` on the third primitive operation now returns the derivatives `[12, 9]` to the second call to `deriveTape` on a primitive operation, which in turn returns them to the first call to `deriveTape` on a primitive operation, which, once more, returns the derivatives. Thus, `deriveTape` terminates with the output `[12, 9]` (bottom right).

Finally, each value of `deriveTape` is packed in a `Primitive`. Thus, `evalVjp` returns `[Primitive(12), Primitive(9)]`. As the partial derivatives of $y \cdot x \cdot x + (2 + 2)$ with respect to x and y respectively are 12 and 9 when $x = 3$ and $y = 2$, this is correct.

—**End of example**———————————————————————————

While illustrating the correctness of this implementation of reverse mode AD, the above example comes close to showing a more unfortunate property of the implementation. As the top right node has two incoming edges, `deriveTape` is called on it twice. Because the node represents a parameter, this did not have big ramifications on the example. If, instead, it represented a primitive operation, the entire computation graph of that operation would have to be traversed once for each incoming edge. This breaks the premise of the time complexity analysis in section 2.1.2, as each node is traversed more than once. In fact, the worst case time complexity of this algorithm is $O(n + o \cdot c^m)$, where o is the number of outputs, c is the maximum number of operands to a primitive operation, n is the total number of operations in the function, and m is the total number of primitive operations in the function. This can be achieved by deriving a function which, using the primitive operation with the maximum number of operands, performs said operation with all operators being equal to a single parameter, and then takes the output of that, and runs the operation again with all parameters equal to the output, and so on. This essentially creates a computation graph, which would be evaluated much like a tree where each node has c children, the depth of the tree being equal to the number of primitive operations in the function.

In order to solve this, memoization could be implemented in `deriveTape`. For this to work, the returned derivatives of the tape would have to be multiplied by their sensitivity after the execution of the function, as the change in sensitivity would otherwise break the cache. This, however, has not yet been implemented due to time constraints.

3.3 Bridging the algorithms, and enabling the calculation of n-th derivatives through nested calls to `jvp` and `vjp`

As illustrated, `jvp` and `vjp` calculate the derivatives of a function by stepping through a trace of its primitive operations, and applying the chain rule to them. Thus far, the calculation of the partial derivatives of primitive operations, as well as the application of the chain rule, have been handled by mathematical expressions, which are hardcoded into the AD implementation. However, these calculations purely consist of primitive operations, and, were they traced, AD could be performed on them. AD can therefore, through nested calls to `jvp` and `vjp`, be used to find higher order derivatives of functions, if the primitive operations performed when AD is calculating a derivative can be traced.

One way of achieving this in the current AD implementation is to calculate the partial derivatives of primitive operations, as well as the applications of the chain rule, using calls to `doOp`. As partial derivatives are calculated using the primal values of their operands, doing so would require representations of values in `JvpValue`s and `VjpValue`s to be extended from scalars to a sum type of `Primitive`s, `JvpValue`s, and `VjpValue`s (henceforth referred to as `ADValue`). In order to find the n-th derivative of a function, it would then have to be nested in n calls to `jvp` or `vjp`, each wrapping its parameters in either a `JvpValue` or a `VjpValue`. In the innermost call, the function would be run, each of its parameters wrapped in n `JvpValue`s or `VjpValue`s. After execution, each call to `jvp` or `vjp` would then strip one `JvpValue` or `VjpValue`, eventually resulting in a `Primitive` of the n-th derivative of the function.

—Example 5

While this approach closely resembles that, which is implemented in the Futhark interpreter, it has one notable issue, which must be considered before it can be implemented. Consider evaluating the following expression:

```
jvp (λx → jvp (λy → x + y) 3 1) 2 1
```

Following the approach explained above, and keeping consistent with the previous implementation of `jvp`, it is clear that `x` would be augmented to contain the value `JvpValue(Primitive(3), Primitive(1))`, while `y` would be augmented to `JvpValue(Primitive(2), Primitive(1))`. Assuming that `Primitive` values are treated as scalars, `doOp` would return `JvpValue(Primitive(5), Primitive(2))`, when evaluating `x + y`. Thus the return value of the innermost call to `jvp` would be `Primitive(2)`. This is clearly not correct, as the derivative of $x + y$ with respect to y is 1.

This issue is also known as *pertubation confusion*, and is a common problem when implementing forward mode AD[4]. The error happens, as the algorithm has no way to distinguish the seed values assigned in the innermost call to `jvp` from those assigned in the outermost call. Thus, the derivative of x with respect to x is treated as if it was the derivative of x with respect to y. A similar issue plaques the implementation of `vjp`, but instead of confusing the seed values, it would confuse the IDs of the variables, thinking that x and y refer to the

same variable.

—**End of example**———————————————————————————————

The solution to this issue is to save a *depth* alongside each `JvpValue` and `VjpValue`. This determines which call to `jvp` or `vjp` the value is associated with. Higher depths means that the call, which generated the `JvpValue` or `VjpValue`, was further down the call stack than those of values with lower depths. By the same logic, having different depths means that values were generated by different calls, and thus that their seed values, or IDs are not comparable.

When operands are of different depths, the ones of lower depths are treated as constants, and wrapped in the value representation of the operand of the highest depth. The resulting value is given the maximum depth of all the operands. In doing so, it is guaranteed that, if a parameter of the innermost call is present in the expression, which is being derived, the returned depth will match that of the innermost call. By the constant rule, the derivative of an expression is known to be `0`, if it returns a value of a lower depth than the innermost call to `jvp` or `vjp`. Otherwise, the depth will be equal to that of the innermost call, and the value will be in the same representation as that which the innermost call augments its parameters with. Thus, the return value of a function called by `jvp` is guaranteed to either have a lower depth than that of the call, in which case the derivative is `0`, or be a `JvpValue`, which can be derived using `deriveJvp`. The same argument can be made for `vjp` and `deriveVjp`s. Note that this only works because the language, in which AD is being implemented, is purely functional, and cannot return closures (functions, which pull certain variables out of scope). If this was not the case, a value could be pulled out of its scope, and used in calculations at lower depths, which would return a value with a higher depth, and thus potentially a different type than that of the call to `jvp` or `vjp` with a lower depth. Futhark can return closures, but it places the limit that the return types of functions passed to `jvp` or `vjp` cannot be functions, nullifying this issue.

The inclusion of a depth also solves a separate issue, which prevented the previous forward mode and reverse mode implementations from coexisting: Applying a primitive operation to a mix of `JvpValue`s and `VjpValue`s is ill-defined. However, since `JvpValue`s and `VjpValue`s cannot exist on the same depth, one would be wrapped in the representation of the other. Thus, after considering the depths of parameters, it is guaranteed that primitive operations are only applied to values of the same representation, making their application well-defined.

Forward mode and reverse mode AD, which can be nested, can now be implemented in the following steps:

1. Change the value representation of `JvpValue`s and `VjpValue`s to add a depth, and replace all scalars with a sum type of `Primitive`s, `JvpValue`s, and `VjpValue`s (`ADValue`).
2. Modify `doOp` to handle depth, and contain a version of its `jvp` and `vjp` enabled logic, which operates on `ADValue`s instead of on scalars.
3. Modify `deriveTape`, `evalJvp`, and `evalVjp` to account for the above changes.

Step 1 Modifying `JvpValue` is simple - the type of its primitive value and derivative must be changed to `ADValue`, and a scalar depth must be added. The new definition of `JvpValue` is `JvpValue(p, d, dep)` where `p` is the primal value, `d` is the derivative, and `dep` is the

depth.

As the primal value of a `VjpValue` is saved in its `Tape`, each constructor of `Tape` must be modified to swap the type of its primitive from a scalar to an `ADValue`. Furthermore, a depth must be added to `VjpValue`. The new definition of `VjpValue` is `VjpValue(tp, dep)` where `tp` is its `Tape`, and `dep` is its depth.

As the definitions of `JvpValue` and `VjpValue` have changed, so must the functions which operate on them. Whereas `getValue(v)` used to return a scalar of the primal value of `v`, it is now less clear what it should return. Instead, two functions can be defined: `getPrimal(v)` which returns the primal value of `v` as an `ADValue`, and `getScalarPrimal(v)` which returns the primal value of `v` as a scalar. They can be defined as follows:

```
1: getPrimal(JvpValue(p, d, dep)): return p
2: getPrimal(VjpValue(tp, dep)):   return getTapeValue(tp)
3: getPrimal(Primitive(v)):        return Primitive(v)

4: getScalarPrimal(Primitive(v)):  return v
5: getScalarPrimal(v):             return getScalarPrimal(getPrimal(v))
```

Given a `JvpValue`, `getPrimal` simply unwraps and returns its primal value (line 1). Given a `VjpValue`, it calls `getTapeValue`, and returns the result (line 2). Note that `getTapeValue` now returns an `ADValue`, and not a scalar, as the type definition of `Tape` has changed. Finally, given a `Primitive`, `getPrimal` returns the primitive itself (line 3). While a `Primitive` does not have a primal value per se, the idea behind primal values is that they hold the value, which would have been held, if the innermost layer of AD was not present. Since `Primitive`s do not store AD information, they remain the same when when running AD. Therefore their regular values are also their primal values.

Whereas `getPrimal` strips the innermost layer of augmented AD data, `getScalarPrimal` must strip every layer to obtain only the primal value with no augmented information. This function will be used to modify the values, which are passed to non-derivable operations, as they are only designed to work with scalars. To achieve this, `getScalarPrimal` recursively calls itself, stripping one layer at a time using `getPrimal` (line 5). When a `Primitive` value is reached, it unwraps and returns its scalar value (line 4).

With the addition of depths, it is useful to define a new function, `getDepth(v)`, which returns the depth of a value. It can be defined as follows:

```
1: getDepth(JvpValue(p, d, dep)): return dep
2: getDepth(VjpValue(tp, dep)):   return dep
3: getDepth(Primitive(v)):        return 0
```

The first two lines are trivial - they simply return the depths of `JvpValue`s and `VjpValue`s respectively. However, for `Primitive`s, the returned depth is `0` (line 3). This is because values of lower depths act as constants for those of higher depths. As `Primitive`s should always act as constants, their depths should be lower than any `JvpValue`s and `VjpValue`s. Note that this requires that the minimum depth given to a `JvpValue` or a `VjpValue` is `1`.

Finally, `augmentJvp` and `augmentVjp` can be updated like so:

```
1: augmentJvp(v,  d, dep): return JvpValue(v, d, dep)
2: augmentVjp(v, id, dep): return VjpValue(TapeVar(id, v), dep)
```

The primal value of a `JvpValue` and a `VjpValue` can now be any `ADValue`. The same is true for the derivatives of `JvpValue`s. Meanwhile, `id` and `dep` are scalars.

Step 2 With the value type of `JvpValue`s and `VjpValue`s changed, `doOp` can now be modified. It can be rewritten recursively, each call updating the augmented information of the innermost layer of AD on the operands, stripping that layer off, and calling itself on the stripped operands. This updates the information on every layer of AD, until the operands are stripped of all AD information, and are thus `Primitive`s. Their scalar values can then be extracted, the operation run, and its result returned as a `Primitive`, ending the recursion. Finally, each call to `doOp` will augment the `Primitive` with its updated AD information.

The updated version of `doOp` is as follows:

```
doOp(op, p_1...p_n):
 1: (mv, md) ← maxBy(getDepth, p_1...p_n)
 2: match type of mv
 3:     with Primitive:
 4:         pv_1...pv_n ← map(getScalarPrimal, p_1...p_n)
 5:         return Primitive(op(pv_1...pv_n))
 6:     with JvpValue:
 7:         p'_1...p'_n ← map(λv → if getDepth(v) = md
                                  then v
                                  else JvpValue(v, 0, md)
                           , p_1...p_n)
 8:         pv ← doOp(op, map(getPrimal, p'_1...p'_n))
 9:         dv ← calculateJvpDerivative(p'_1...p'_n)
10:         return JvpValue(pv, dv, md)
11:     with VjpValue:
12:         p'_1...p'_n ← map(λv → if getDepth(v) = md
                                  then v
                                  else VjpValue(TapeConst(v), md)
                           , p_1...p_n)
13:         pv ← doOp(op, map(getPrimal, p'_1...p'_n))
14:         tp ← updateVjpTape(op, p'_1...p'_n, pv)
15:         return VjpValue(tp, md)
```

First, the operand with the maximum depth (`mv`), as well as its depth (`md`) are found (line 1). If several operands share the same depth, any one of them can be chosen. The type of `mv` now determines the type of the outermost layer of AD, which is currently present in the operands. Thus, the return type of `doOp` must match the type and depth of `mv`. If `mv` is a `Primitive`, is it guaranteed that every operand is a `Primitive`, as no operand has a higher depth than `0`. In this case, the scalar value of each operand is extracted with `getScalarPrimal` (line 4), and the primitive operation can be applied to the scalar values, the result returned as a `Primitive` (line 5). If AD is not running, this is the only case, which the code will reach.

If, instead, `mv` is a `JvpValue` or a `VjpValue`, it means the output of the function must be augmented with the AD information, which is appropriate to the respective type. In order to do this, every operand must be updated to have the same representation as `mv`. If the depth of the operand is equal to that of `mv`, it need not be modified, as it already has the

correct representation. Otherwise, it must be represented as a constant would for `jvp` or `vjp`, depending on the type of `mv`. A `JvpValue` is a constant, if its derivative is `0` (line 7), and a `VjpValue` is a constant, if its `Tape` is a `TapeConst` (line 12).

In order to calculate the value of the previous layer of AD (i.e. the primal value of the current layer of AD), each operand must be stripped of its outermost layer of AD information by using `getPrimal`. The stripped operands can now be passed to `doOp` along with `op`. This handles every layer of AD beneath that of `mv` (lines 8 and 13).

The AD information of the innermost layer of AD can now be calculated. As the function is already quite long, this is handled by two new functions, `calculateJvpDerivative(`$\texttt{p}_1 \ldots \texttt{p}_n$`)` and `updateVjpTape(`$\texttt{p}_1 \ldots \texttt{p}_n$`, p)`, which calculate the derivative of the resulting `JvpValue`, or the tape of the resulting `VjpValue` respectively (lines 9 and 14). These functions will be described in the following text.

Finally, a `JvpValue` or `VjpValue` can be assembled, and returned (lines 10 and 15). Note that the returned value is given the same depth as `mv`.

`calculateJvpDerivative` and `updateVjpTape` will now be defined. Note that, in order to avoid endless recursion, they must never pass a value of the highest depth (such as one of the parameters, $\texttt{p}_1 \ldots \texttt{p}_n$) to `doOp`. This will not be an issue for the implementations below.

`calculateJvpDerivative` can be defined as follows:

`calculateJvpDerivative(`$\texttt{p}_1 \ldots \texttt{p}_n$`):`

1: $\texttt{pd}_1 \ldots \texttt{pd}_n \leftarrow$ `getPartials(op)`

2: $\texttt{pv}_1 \ldots \texttt{pv}_n \leftarrow$ `map(getPrimal,` $\texttt{p}_1 \ldots \texttt{p}_n$`)`

3: $\texttt{dv}_1 \ldots \texttt{dv}_n \leftarrow$ `map(deriveJvp,` $\texttt{p}_1 \ldots \texttt{p}_n$`)`

4: **return** $\texttt{eval}(\texttt{dv}_1 \cdot \texttt{pd}_1(\texttt{pv}_1 \ldots \texttt{pv}_n) + \ldots + \texttt{dv}_n \cdot \texttt{pd}_n(\texttt{pv}_1 \ldots \texttt{pv}_n))$

The code is nearly identical to that of the second branch of `doOp` in the implementation of forward mode, however it is functionally different. As in the original implementation, the function starts by retrieving the partial derivatives of `op` (line 1). It then extracts the primal values (line 2), as well as the derivatives (line 3) of the operands. Note that `deriveJvp` now returns an `ADValue` instead of a scalar, due to the modification of `JvpValue`. Both the primal values and the derivatives will be of a lower depth than the operands, as values cannot contain values of an equal or higher depth. The biggest difference between the old implementation and this one is that the application of the chain rule, as well as the computation of the partial derivatives, is run through the interpreter (line 4). This is essentially syntactic sugar meaning that every primitive operation of the calculation is run through `doOp`. The result is a `JvpValue` of the same depth as the deepest primal value or derivative of the operands. Note that this will still be of a lower depth than the operands. Finally, the function returns the calculated derivative (line 4).

`updateVjpTape` can be defined as follows:

`updateVjpTape(op,` $\texttt{p}_1 \ldots \texttt{p}_n$`, pv):`

1: $\texttt{tp}_1 \ldots \texttt{tp}_n \leftarrow \texttt{map}(\lambda(\texttt{VjpValue(tp, dep)}) \rightarrow \texttt{tp},\ \texttt{p}_1 \ldots \texttt{p}_n)$

2: **return** $\texttt{TapeOp(op, tp}_1 \ldots \texttt{tp}_n\texttt{, pv)}$

As every operand is guaranteed to be a `VjpValue`, the function must simply unwrap the tape of the operands (line 1), and return a `TapeOp` with the operator, the tapes, and the primal value, which was calculated in `doOp`.

Step 3 Much like `calculateJvpDerivative`, the only change which must be made to `deriveTape` is running its calculations through `eval`. These include the calculation of its sensitivity (when deriving a primitive operation; line 10), and the addition of sensitivities (when deriving a variable; line 3). Doing so results in the following code:

```
deriveTape(tp, s, dv_1...dv_m):
 1: match tp
 2:     with TapeVar(id, v):
 3:         dv_id ← eval(dv_id + s)
 4:         return dv_1...dv_m
 5:     with TapeConst(v): return dv_1...dv_m
 6:     with TapeOp(op, tp_1...tp_n, v):
 7:         pv_1...pv_n ← map(λtp → getTapeValue(tp), tp_1...tp_n)
 8:         pd_1...pd_n ← getPartials(op)
 9:         pdv_1...pdv_n ← map(λpd → eval(s · pd(pv_1...pv_n)), pd_1...pd_n)
10:         dv_1...dv_m ← reduce(λ(dv_1...dv_m, tp, pdv) →
                                    deriveTape(tp, pdv, dv_1...dv_m)
                              , dv_1...dv_m
                              , zip(tp_1...tp_n, pdv_1...pdv_n))
11:         return dv_1...dv_m
```

`evalJvp` and `evalVjp` can now be modified as follows:

```
evalJvp(f, v_1...v_n, s_1...s_n):
 1: dep ← getDepth()
    ▷ Step 1: Augment each input value with its respective seed value
 2: v'_1...v'_n ← map(λ(v, s) → augmentJvp(v, s, dep)
                      , zip(v_1...v_n, s_1...s_n)
    ▷ Step 2: Evaluate the function (through the interpreter) on the augmented inputs
 3: o_1...o_m ← eval(f(v'_1...v'_n))
    ▷ Step 3: Extract the derivative of each output
 4: return map(deriveJvp, o_1...o_m)

evalVjp(f, v_1...v_n, s_1...s_m):
 1: dep ← getDepth()
    ▷ Step 1: Augment each input value
 2: v'_1...v'_n ← map(λ(v, id) → augmentVjp(v, id, dep)
                      , enumerate(v_1...v_n))
    ▷ Step 2: Evaluate the function (through the interpreter) on the augmented inputs
 3: o_1...o_m ← eval(f(v'_1...v'_n))
    ▷ Step 3: Derive the outputs of the function
 4: return join(λ(a, b) → eval(a + b)
               , map(λ(o, s) → deriveVjp(o, s, n)
                     , zip(o_1...o_m, s_1...s_m)))
```

Both functions must now initially get a depth (line 1 of both functions). This can be implemented in many ways - for example by taking the length of the call stack. The augmentation functions are run as they were before, but they must now also take the depth (line 2 of both functions). The function is then evaluated with the augmented parameters, as it was before (line 3 of both functions). Finally, the outputs are derived and returned (line 4 of both functions). There are two changes to the code on line 4: Firstly, the return values of the `derive` functions are now `ADValue`s, so they need not be wrapped in `Primitive`s. Secondly, in `evalJvp`, the addition operation in the `join` function is wrapped in `eval`. Note that the second branch of `deriveVjp` must be modified to return `Primitive`s instead of scalars, when the seed of an output is `0`.

3.4 Bringing the algorithm to the Futhark interpreter

To aid the explanation of the AD implementation in the above sections, the language, for which it was implemented, was highly simplified. Futhark, however, has typed primitives, and composite types, and it lacks definitions of `doOp`, and `eval`, which are as versatile as those used in the pseudo code. The following section will briefly mention some of the previously unaddressed decisions and challenges in implementing AD for the Futhark interpreter.

Implementing `ADValue`

As mentioned in section 2.2.1, Futhark already stores its values in a sum type, one of its constructors, `ValuePrim`, representing all primitive values. While it may seem intuitive to replace the scalar value in that constructor with an `ADValue`, as these, too, can store `Primitive`s, it was ultimately decided against, so as to reduce the impact of the AD implementation on the runtime of programs, which do not use AD. Instead, a separate constructor, `ValueSeed`, was made to store either `JvpValue`s or `VjpValue`s. If a primitive operation is applied only to `ValuePrim`s, the operation is handled by the preexisting code in the interpreter. If, instead, one or more of the operators is a `ValueSeed`, it is passed to `doOp`, and handled by the AD implementation.

Implementing `doOp`

Whereas every primitive operation goes through `doOp` before AD is implemented in the simplified interpreter, the logic for calling and applying primitive operations was initially split over many parts of the Futhark interpreter. In order to bring these functions together, `doOp` was implemented. The Futhark interpreter initially had a few types, which were used to differentiate primitive operations. These essentially grouped the operations into four categories, each contained in a different type:

- `BinOp`, which stores binary arithmetic operations,
- `CmpOp`, which stores binary comparison operations,
- `UnOp`, which stores unary operations, and
- `ConvOp`, which stores operations, which convert a `ValuePrim` from one type to another.

Furthermore, Futhark has built in mathematical functions, which may take any number of parameters. These are differentiated by their names, which are stored as strings.

The first step in implementing `doOp` was to create a sum type, `Op`, to hold any type of operation, including functions. This is essentially the `op` parameter of `doOp`, save for the fact that it cannot be applied as a function.

The parts of the code, which were responsible for applying primitive operations could now be directed to `doOp`. As each part had its own logic for type checking operators, and choosing the appropriate version of a primitive operation to apply (such as single and double precision floating point operations), this, too, had to be implemented in `doOp`. Finally, `doOp` applies the primitive operation. A minor difference between the implementation in the Futhark interpreter, and that in section 3.3 is that in Futhark, the calculation of the outermost `Primitive` value is calculated in `doOp`, while the info for the remaining layers of AD is handled in a separate function called `handleOp`. This is purely a style choice to keep the functions shorter and more focused.

Updating the different layers of the AD information works much like it did in section 3.3, with one major difference: The calculation of partial derivatives is not run through the interpreter.

Calculating the partial derivatives without `eval`

AD has already been implemented in the Futhark compiler, and the code base therefore has a list of partial derivatives. As these must be calculated in the program, which is output by the compiler, the derivatives are not implemented as functions, but rather in the form of syntax trees containing solely mathematical operations. As these are all primitive operations they can be run through `doOp`. Thus, the primitive operations are essentially calculated in the AD implementation of the Futhark interpreter by running their syntax trees through a tiny interpreter (a function called `runPrimExp`), which offloads the application of all primitive operations to `doOp`, effectively recursively calling itself, when AD is being used to calculate higher order derivatives. The chain rule is also applied using `doOp`, however without using `runPrimExp`. Note that each call to `doOp` must pass an operation, which matches the type of the operands. This is simply achieved by matching the operation to the type of the operands. `deriveTape` (called `deriveVjp` in the Futhark interpreter) is implemented in a similar manner.

Implementing `evalJvp` and `evalVjp`

As Futhark has composite values, and can pass them to `evalJvp` and `evalVjp` (called `jvp2` and `vjp2` in the code), its implementation of these functions is considerably more complex. If composite values are passed to `jvp2`, the structure of the value holding the primal values and that of the value which holds the seeds must be identical. The primal values and the seeds are then be paired up, and traversed, returning a composite type with an identical structure, holding the augmented values. The same is true for the seed values, and the outputs of a function in `vjp2`. This is achieved by performing an in-place replacement of the primal values with their augmented counterparts. In `vjp2`, this must be done while maintaining an incrementing value, which is used to generate the unique ID for each variable. Furthermore, as `vjp2` must return the value of the partial derivative of each parameter in the same type as the parameter, it must collect the types of the parameters, when their values are augmented. When the output values are joined together and returned, it must then represent constants as the type of their respective parameter.

Type conversion of `ADValues`

As `ValuePrim`s in Futhark support type conversion, so must `ADValue`s. In forward mode AD, this simply entails converting the primal value as well as its derivative to the desired type. In reverse mode, the initial value of a sensitivity (the seed value) for each output must match the primal value of the output, such that it matches the primal values of the operands of the final primitive operations in its computation graph. However, if a type conversion occurs for an `ADValue`, it must be performed in reverse, when moving up the computation graph, such that the sensitivity continues to match the types of the operands for the consequent primitive operations. Type conversions must therefore be stored in the `Tape` just like any other primitive operation.

4 Testing and benchmarking

The implementation can now be tested for correctness, and benchmarked.

4.1 Correctness

As the Futhark compiler already has an AD implementation, a series of relevant tests exist in the code base. These include tests of various primitive operations, control flow, higher order function calls, type conversions, and previous implementation issues for forward mode, and reverse mode AD.

Running the Futhark interpreter through the full AD test suite reveals that 11 / 151 test cases are simply so intensive that they take too long, or too much memory, for the interpreter to run. Some of these are benchmarks, while others are real world examples. Disabling these tests reveals that 136 / 140 remaining tests pass. In order to know if there truly are issues with the AD implementation, the failing tests will be examined:

- `cmp0.fut`. In this test, the interpreter is being asked to derive $a = b$ with respect to a and b using forward mode and reverse mode AD. If the two numbers are equal, the derivative must be *true* or 1, and otherwise $false$ or 0. However, this is not derivable when $a = b$, and so the value of the actual derivative is undefined. Therefore, the AD implementation for the Futhark interpreter returns 0.

- `iota0.fut`. In this case, the sum of `iota` must be equal to the number of elements in it, if the number of elements is defined by the variable, which the partial derivative is taken with respect to. The AD implementation in the Futhark interpreter does not consider `iota` a primitive operation, as it does not output a single scalar, and, even if it did, the value of each scalar is not dependent on the input to `iota`. The scalars output by `iota` are therefore considered constants.

- `reducebyindex5.fut`. This test covers the derivative of a second order function called `reduce_by_index`. The function acts as `reduce`, however it only reduces certain elements in the input list, given by a list of indexes. Interestingly, when running the Futhark compiler on `reducebyindex5.fut`, `reduce_by_index` will provably return the value of a parameter (thus not a constant), yet performing reverse mode AD returns a partial derivative of `0` for said parameter. While this result seems wrong, the test expects this output. The interpreter implementation outputs `1`, as should be expected. The maintainer of Futhark has been notified of this, however, no response has been given at the time of writing. Whether this result is due to an error in the compiler and the expected output in the test, or the test gives an invalid input to `reduce_by_index`, or there is a flaw in this analysis is therefore unknown for the time being.

- Due to time constaints, an analysis of `reducebyindex1.fut` is still pending, however the possible flaw in `reducebyindex5.fut` brings the validity of this test into question. It should also be noted that there are six test cases for `reduce_by_index`, the remaining four passing.

As the former two failing tests clearly test quirks in the AD implementation of the Futhark compiler, and **not** whether or not the AD implementation returns the correct derivative when given a derivable function, they can be said to be irrelevant in testing the correctness of an AD implementation. The third test is likely flawed, and the validity of the fourth is

questionable, given that it tests the same function as the third. Testing has therefore likely not found any errors in the AD implementation of the Futhark interpreter.

4.2 Benchmarking

In benchmarking the AD implementation, two queries were of interest: How much slower a program runs when applying AD, as opposed to running it normally; and whether or not the existence of the AD implementation slows down a program when AD is not applied. Testing the former requires timing the execution of a program when applying no AD, forward mode AD, and reverse mode AD, and calculating the factor of which performing AD increases the runtime of the program. The latter can be tested by running a program without AD using the interpreter before and after AD support was added.

In order to test these queries, a Futhark program was written to provide a workload of ample size to benchmark the implementation. The following is the program used in benchmarking:

```
def f (x: f32) =
  loop x = x for i < 100000 do
    let s = (i32.f32 (x * 10)) %% 4 in
    if x > 100 then
      if s == 0 then  1 + f32.sin x  else
      if s == 1 then  1 + f32.cos x  else
      if s == 2 then  f32.log1p x    else
      if s == 3 then  f32.sqrt x     else
      x / 13
    else
      if s == 0 then  x + 10                  else
      if s == 1 then  x ** 3                  else
      if s == 2 then  f32.exp (x / 10)        else
      if s == 3 then  reduce (*) x [2, x, 5]  else
      x * 1.3

def main (d: i32) (x: f32) =
  if d == 0      then f x
  else if d == 1 then jvp f x 1
  else if d == 2 then vjp f x 1
  else 0
```

It consists of a function, `f`, which takes a single floating-point parameter, `x`, and modifies it through a process using various primitive operations, control flow, and second order functions, before finally returning it. The modification of `x` is performed in a loop, the number of iterations determining the amount of work of the benchmark.

The entry function, `main`, takes two parameters: `d`, which determines whether to perform no AD (0), forward mode AD (1), reverse mode AD (2), or returning 0 without running the function (3); and `x`, which determines the initial value of `x` in `f`.

When the Futhark interpreter runs a program, it must first perform lexing, parsing, and type checking. This adds a constant amount of time to each of the rest runs, pushing the factor of execution time with vs. without AD towards 1. As only the slowdown of evaluating a function with AD is desired, this constant factor must be removed. Benchmarks are therefore

performed not only for no AD, forward mode AD, and reverse mode AD, but also for the branch, which returns 0. The execution time of the latter benchmark is then subtracted from those of the former three to obtain a time, which better represents only the evaluation of the function.

The benchmarks were performed using `hyperfine` with 5 warmup runs followed by 50 runs. The initial value of `x` was 3. The results are as follows:

Branch	Execution time	Evaluation time	Factor vs. no AD
Constant (3)	175 ms ± 7 ms	0 ms	N/A
No AD (0)	871 ms ± 14 ms	696 ms	1.000
Forward mode AD (1)	2008 ms ± 26 ms	1833 ms	2.634
Reverse mode AD (2)	1637 ms ± 21 ms	1462 ms	2.101

Figure 4: Benchmarks of the implementation when performing forward mode and reverse mode AD. It contains the execution time of a constant function (3), the benchmarked function without applying AD (0), the benchmarked function when applying forward mode AD (1), and the benchmarked function when applying reverse mode AD (2). For each benchmark, data is included on the time it took to lex, parse, and evaluate; an estimate of the time it took to evaluate; and the evaluation time estimate divided by the evaluation time estimate when performing no AD (0).

Version	Execution time (0)	Constant time (3)	Evaluation time	Factor vs. before AD
Before AD	871 ms ± 14 ms	175 ms ± 7 ms	696 ms	1.000
After AD	867 ms ± 17 ms	175 ms ± 8 ms	692 ms	0.994

Figure 5: Benchmarks of the implementation when not performing AD. It contains the execution time of the benchmarked function without running AD on the interpreter before, and after AD was added. For each benchmark, data is included on the time it took to lex, parse, and evaluate the benchmarked function; the time ot took to lex, parse, and evaluate the constant function; an estimate of the time it took to evaluate the benchmarked function, produced from the two former results; and the evaluation time estimate divided by the evaluation time estimate when when running the version of the Futhark interpreter before AD was added.

The results indicate no increase of evaluation time of programs, which do not apply AD after modifying the interpreter (figure 5). In fact, the results show an insignificant decrease in computation time, however this is likely due to a small inaccuracy in the time measurement. This is not surprising, as the value type of the Futhark interpreter was already a sum type. In fact, no new match cases have been added to any part of the code, which runs when AD is not applied, so assuming that sum types are stored as enumerations in Haskell, there should be no increase in the number of operations performed by the modified interpreter.

There is an approximate slowdown of factor 2.634 when performing forward mode AD, and factor 2.101 when performing reverse mode AD (figure 4). Forward mode is most likely slower, as it, unlike reverse mode, calculates the derivatives of `s`, which are not part of the computation graph of the output. Given that the calculation of a partial derivative, and the application of the chain rule tends to make the calculation of a derivative more intensive

than that of its primal value, this degree of slowdown can be considered small, however there are a few important caveats:

Caveat #1 As this implementation of AD runs in an interpreter, there is a lot of overhead in evaluating programs. While the main factor contributing to the speed of a program may still be the application of its primitive operations, the extra overhead of traversing the syntax tree of a function, and handling value types adds a constant overhead to the execution of a function, with or without AD. This sways each factor closer to 1.

Caveat #2 More importantly, this is only a benchmark of a single program. While this program was in no way designed to play into the strengths of the current implementation, it may not be representative of the broad array of cases for which AD is used. As stated at the end of section 3.2, a program could easily be designed to tank the execution time of the current implementation of reverse mode AD. In fact, this could be done by taking the sum of every value in the loop.

5 Conclusion

AD was successfully implemented in Futhark, and the implementation passed 136 / 138 relevant tests in the Futhark AD test suite. Of the failing tests, one is seemingly flawed, while the validity of the second is questionable.

The implementation is able to perform both forward mode and reverse mode AD, and it supports nesting the AD functions to calculate higher order derivatives. While both modes performed well in benchmarks, the reverse mode AD implementation is suboptimal as its performance can tank, if it is run on programs with primitive operations, where many of the operands refer to the same value.

5.1 Future work

Before this implementation can be adopted into the Futhark code base, a few important issues must be solved:

The suboptimal implementation of reverse mode AD The implementation of reverse mode AD must be optimized to avoid performing duplicate work. This might be solved by taking a dynamic programming approach, uniquely tagging every `Tape`, and performing memoization on each call to `deriveTape`.

Compatibility with the Futhark compiler Some derivatives produce differing results in the Futhark compiler, and in the implementation for the Futhark interpreter. While these derivatives are all of non-derivable operations, making the correct answer undefined, it should be considered whether or not the compiler and the interpreter must produce the same result, and, if so, what the result should be.

The state of the code base Due to time constraints, the code base for the AD implementation is fairly messy. Before merging it into the official Futhark code base, it would have to be cleaned up, and made to follow the style of the existing code more closely. Certain functions, such as those for type checking and conversion, can possibly also be replaced by ones, which already exist in the code base.

Unifying the value representations, and the primitive operation calls? As the AD implementation adds a new value type, which simply wraps around primitive values (`Primitive`), and adds a function, which applies primitive operations to said values (`doOp`) it may be possible to remove the old primitive operation application logic, and run solely off the new logic. This would reduce the complexity of the code, but require a rewrite of certain parts of the interpreter. It may be interesting to explore, if this is a viable approach.

6 References

[1] Henriksen, T. (2017) Design and implementation of the Futhark programming language. thesis. University of Copenhagen, Faculty of Science Department of Computer Science.

[2] Baydin, A.G. et al. (2018) 'Automatic Differentiation in Machine Learning: a Survey', Journal of Machine Learning Research

[3] diku-dk/futhark | A data-parallel functional programming language GitHub. Available at: `https://github.com/diku-dk/futhark` (Accessed: 11 August 2024).

[4] MANZYUK, O. et al. (2019) 'Perturbation confusion in forward automatic differentiation of higher-order functions', Journal of Functional Programming, 29. doi:10.1017/s095679681900008x.

[5] Forward-mode automatic differentiation Futhark-Lang.org. Available at: `https://futhark-lang.org/examples/forward-ad.html` (Accessed: 11 August 2024).

[6] Reverse-mode automatic differentiation Futhark-Lang.org. Available at: `https://futhark-lang.org/examples/reverse-ad.html` (Accessed: 11 August 2024).