Two Things I Did:
Parallel Differentiation and Rank Polymorphism

Robert Schenck

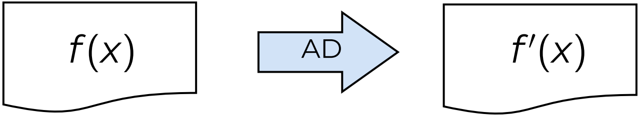
March 18th, 2025

Overview

- This talk is about two things I did during my PhD studies:

Overview

- This talk is about two things I did during my PhD studies:
  - Thing #1: parallel automatic differentiation:



Overview

- This talk is about two things I did during my PhD studies:
  - Thing #1: parallel automatic differentiation:

$$f(x) \xrightarrow{\text{AD}} f'(x)$$

  - Thing #2: (pseudo-)rank polymorphism in a statically-typed language:

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} + 5 \xrightarrow{\text{AUTOMAP}} \textbf{map}\ (\textbf{map}\ (+))\ \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}\ \begin{bmatrix} 5 & 5 \\ 5 & 5 \end{bmatrix}$$

Overview

- This talk is about two things I did during my PhD studies:
  - Thing #1: parallel automatic differentiation:

$$f(x) \xrightarrow{\text{AD}} f'(x)$$

  - Thing #2: (pseudo-)rank polymorphism in a statically-typed language:

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} + 5 \xrightarrow{\text{AUTOMAP}} \textbf{map}\ (\textbf{map}\ (+))\ \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}\ \begin{bmatrix} 5 & 5 \\ 5 & 5 \end{bmatrix}$$

- How are these things related?

Overview

- This talk is about two things I did during my PhD studies:
  - Thing #1: parallel automatic differentiation:

$$f(x) \xrightarrow{\text{AD}} f'(x)$$

  - Thing #2: (pseudo-)rank polymorphism in a statically-typed language:

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} + 5 \xrightarrow{\text{AUTOMAP}} \textbf{map}\ (\textbf{map}\ (+))\ \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}\ \begin{bmatrix} 5 & 5 \\ 5 & 5 \end{bmatrix}$$

- How are these things related?
  - Well, they are and aren't—both features belong in scientific programming languages.

Overview

- This talk is about two things I did during my PhD studies:
  - Thing #1: parallel automatic differentiation:

$$f(x) \xrightarrow{\text{AD}} f'(x)$$

  - Thing #2: (pseudo-)rank polymorphism in a statically-typed language:

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} + 5 \xrightarrow{\text{AUTOMAP}} \textbf{map}\ (\textbf{map}\ (+))\ \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}\ \begin{bmatrix} 5 & 5 \\ 5 & 5 \end{bmatrix}$$

- How are these things related?
  - Well, they are and aren't—both features belong in scientific programming languages.
  - Scientific programming is about adopting mathematics for (efficient) computation with a computer.

Scientific Programming

- Desirables for a scientific programming language:

Scientific Programming

- Desirables for a scientific programming language:
  - **High-level of abstraction:**
    - Should support the writing of mathematics as programs easily.
    - Program transformations/optimizations should be simple to express.

Scientific Programming

- Desirables for a scientific programming language:
  - **High-level of abstraction:**
    - Should support the writing of mathematics as programs easily.
    - Program transformations/optimizations should be simple to express.
  - **Principled:**
    - Programming model should follow simple rules and be unsurprising.

Scientific Programming

- Desirables for a scientific programming language:
  - **High-level of abstraction:**
    - Should support the writing of mathematics as programs easily.
    - Program transformations/optimizations should be simple to express.
  - **Principled:**
    - Programming model should follow simple rules and be unsurprising.
  - **Fast:**
    - Gotta go fast!!

Scientific Programming

- Desirables for a scientific programming language:
  - **High-level of abstraction:**
    - Should support the writing of mathematics as programs easily.
    - Program transformations/optimizations should be simple to express.
  - **Principled:**
    - Programming model should follow simple rules and be unsurprising.
  - **Fast:**
    - Gotta go fast!!

	FORTRAN/C	APL	NumPy	?
High-level	✗	✓	✓	✓
Principled	✗	✓	✗	✓
Fast	✓	?	?	✓

Futhark

- Both things (i.e., Thing #1 and Thing #2) were implemented as extensions to the Futhark programming language.

Futhark

- Both things (i.e., Thing #1 and Thing #2) were implemented as extensions to the Futhark programming language.
- Futhark is...

Futhark

- Both things (i.e., Thing #1 and Thing #2) were implemented as extensions to the Futhark programming language.
- Futhark is...
  - Statically typed (with parametric polymorphism):

$$\mathit{id} : \alpha \to \alpha = \lambda x . x$$

Futhark

- Both things (i.e., Thing #1 and Thing #2) were implemented as extensions to the Futhark programming language.
- Futhark is...
  - Statically typed (with parametric polymorphism):

$$id : \alpha \to \alpha = \lambda x . x$$

  - A data-parallel functional array language.
    - Uses a library of parallel operators to build parallel-by-construction programs: **`map`**, **`reduce`**, **`scan`**, **`hist`**, ...

Futhark

- Both things (i.e., Thing #1 and Thing #2) were implemented as extensions to the Futhark programming language.
- Futhark is...
  - Statically typed (with parametric polymorphism):

$$\mathit{id} : \alpha \to \alpha = \lambda x.x$$

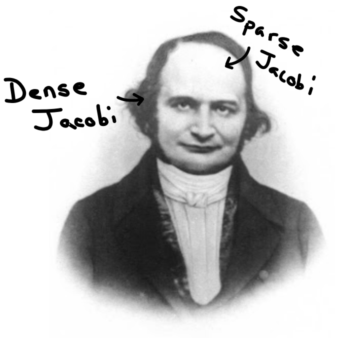
  - A data-parallel functional array language.
    - Uses a library of parallel operators to build parallel-by-construction programs: **`map`**, **`reduce`**, **`scan`**, **`hist`**, ...

Example

```
def dotprod x y = reduce (+) 0 (map (*) x y)
```

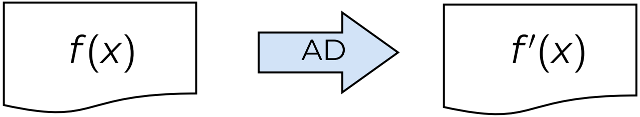
Thing #1:
Parallel Automatic Differentiation

Robert Schenck, Ola Rønning, Troels Henriksen, Cosmin E. Oancea



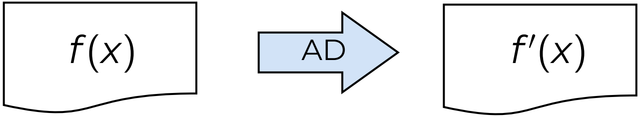
Overview

- **Automatic differentiation (AD)** is a program transformation for differentiation.



Overview

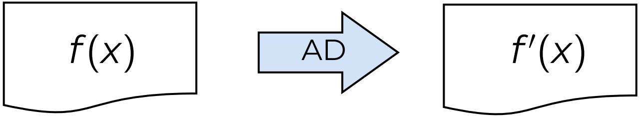
- **Automatic differentiation (AD)** is a program transformation for differentiation.



- Considering AD for a **functional**, **high-level**, and **nested-parallel** array language.

Overview

- **Automatic differentiation (AD)** is a program transformation for differentiation.



- Considering AD for a **functional**, **high-level**, and **nested-parallel** array language.
- All parallelism is made explicit via **parallel combinators**—**`map`**, **`reduce`**, **`scan`**, etc.

High-level AD

Key idea #1: High-level AD

Parallel constructs are differentiated at a **high-level**.

High-level AD

Key idea #1: High-level AD

Parallel constructs are differentiated at a **high-level**.

- Parallel combinators are differentiated with **specialized rewrite rules**.

$$\mathbf{\texttt{map}} \underset{\text{AD}}{\Longrightarrow} \mathbf{\texttt{reduce}} \circ \mathbf{\texttt{map}}, \qquad \mathbf{\texttt{reduce}} \underset{\text{AD}}{\Longrightarrow} \mathbf{\texttt{map}} \circ \mathbf{\texttt{scan}}$$

High-level AD

Key idea #1: High-level AD

Parallel constructs are differentiated at a **high-level**.

- Parallel combinators are differentiated with **specialized rewrite rules**.

$$\mathbf{\texttt{map}} \xRightarrow[\text{AD}]{} \mathbf{\texttt{reduce}} \circ \mathbf{\texttt{map}}, \qquad \mathbf{\texttt{reduce}} \xRightarrow[\text{AD}]{} \mathbf{\texttt{map}} \circ \mathbf{\texttt{scan}}$$

- Differentiated programs benefit from entire optimization pipeline in the compiler.

High-level AD

Key idea #1: High-level AD

Parallel constructs are differentiated at a **high-level**.

- Parallel combinators are differentiated with **specialized rewrite rules**.

$$\texttt{map} \xRightarrow[\text{AD}]{} \texttt{reduce} \circ \texttt{map}, \qquad \texttt{reduce} \xRightarrow[\text{AD}]{} \texttt{map} \circ \texttt{scan}$$

- Differentiated programs benefit from entire optimization pipeline in the compiler.
- Differentiation occurs **before** parallelism is mapped to hardware.

The Tape

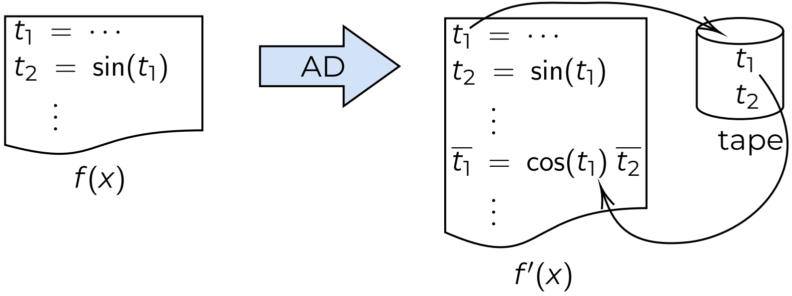
- Variables of the original program appear in the differentiated program.

The Tape

- Variables of the original program appear in the differentiated program.
- All intermediate variables in the original program must be accessible in the differentiated program.

The Tape

- Variables of the original program appear in the differentiated program.
- All intermediate variables in the original program must be accessible in the differentiated program.
- In classic AD, these variables are stored on a dynamically allocated **tape**.



AD by Re-execution

- In our **nested-parallel** context, the tape is **complex/irregular** and must be passed across **deeply nested scopes**. Challenging to implement efficiently.

AD by Re-execution

- In our **nested-parallel** context, the tape is **complex/irregular** and must be passed across **deeply nested scopes**. Challenging to implement efficiently.

Key idea #2: Re-execution

Instead of storing intermediate variables, re-compute them by **re-execution**.

- A classic **space-time tradeoff**.

AD by Re-execution

- In our **nested-parallel** context, the tape is **complex/irregular** and must be passed across **deeply nested scopes**. Challenging to implement efficiently.

Key idea #2: Re-execution

Instead of storing intermediate variables, re-compute them by **re-execution**.

- A classic **space-time tradeoff**.
- **Asymptotics-preserving**: re-execution overhead is a constant for non-recursive programs.

AD by Re-execution

- In our **nested-parallel** context, the tape is **complex/irregular** and must be passed across **deeply nested scopes**. Challenging to implement efficiently.

Key idea #2: Re-execution

Instead of storing intermediate variables, re-compute them by **re-execution**.

- A classic **space-time tradeoff**.
- **Asymptotics-preserving**: re-execution overhead is a constant for non-recursive programs.
- Pretty fast in practice!

Related Parallel AD Work

- **PyTorch, JAX, etc:** Restricted parallel DSLs; AD on fixed set of array primitives.

Related Parallel AD Work

- **PyTorch, JAX, etc:** Restricted parallel DSLs; AD on fixed set of array primitives.

- **Enzyme:** LLVM compiler plugin that does AD on a post-optimization, low-level representation.

Related Parallel AD Work

- **PyTorch, JAX, etc:** Restricted parallel DSLs; AD on fixed set of array primitives.
- **Enzyme:** LLVM compiler plugin that does AD on a post-optimization, low-level representation.
- **Dex:** High-level AD that uses multiple tapes; hard to implement efficiently.

Related Parallel AD Work

- **PyTorch, JAX, etc:** Restricted parallel DSLs; AD on fixed set of array primitives.
- **Enzyme:** LLVM compiler plugin that does AD on a post-optimization, low-level representation.
- **Dex:** High-level AD that uses multiple tapes; hard to implement efficiently.

	PyTorch, JAX, etc.	Enzyme	Dex	?
High-level	✓	✗	✓	✓
Principled	✗	✗	✓	✓
Fast	?	✓	?	✓

A Very Short Introduction to AD

Introduction to AD

- **Goal:** compute the sensitivity of the output y to its inputs x_0, x_1.

$$
\begin{array}{l}
P(x_0,\ x_1):\\
\quad t_0 = \sin(x_0)\\
\quad t_1 = x_1 \cdot t_0\\
\quad y = x_0 + t_1\\
\quad \textbf{return } y
\end{array}
\quad \Longrightarrow \quad
\begin{array}{l}
P'(x_0,\ x_1, \bar{y}):\\
\\
\qquad ?\\
\\
\\
\end{array}
$$

Introduction to AD

$P(x_0,\ x_1):$
$t_0 = \sin(x_0)$
$t_1 = x_1 \cdot t_0$
$y = x_0 + t_1$
return y

$\Longrightarrow$

$P'(x_0,\ x_1, \overline{y}):$

?

- **Goal:** compute the sensitivity of the output y to its inputs x_0, x_1.

Adjoint of a variable

$$\overline{v} \equiv \frac{\partial y}{\partial v}$$

The **sensitivity** of the output y to v.

Introduction to AD

- **Goal:** compute $\overline{x_0}$ and $\overline{x_1}$

$P(x_0,\ x_1):$
$t_0 = \sin(x_0)$
$t_1 = x_1 \cdot t_0$
$y = x_0 + t_1$
return y

$\Longrightarrow$

$P'(x_0,\ x_1, \overline{y}):$

?

return $\overline{x_0},\ \overline{x_1}$

Adjoint of a variable

$$\overline{v} \equiv \frac{\partial y}{\partial v}$$

The **sensitivity** of the output y to v.

Introduction to AD

$$
\begin{array}{l}
P(x_0,\ x_1):\\
\quad t_0 = \sin(x_0)\\
\quad t_1 = x_1 \cdot t_0\\
\quad y = x_0 + t_1\\
\quad \textbf{return } y
\end{array}
\quad \Longrightarrow \quad
\begin{array}{l}
P'(x_0,\ x_1, \overline{y}):\\
\quad t_0 = \sin(x_0)\\
\quad t_1 = x_1 \cdot t_0\\
\quad y = x_0 + t_1\\
\\
\\
\\
\\
\quad \textbf{return } \overline{x_0},\ \overline{x_1}
\end{array}
$$

- **Goal:** compute $\overline{x_0}$ and $\overline{x_1}$

Adjoint of a variable

$$\overline{v} \equiv \frac{\partial y}{\partial v}$$

The **sensitivity** of the output y to v.

- Adjoints depend on primal values. Add the statements of the original program.

Introduction to AD

$$P(x_0,\ x_1):$$
$$t_0 = \sin(x_0)$$
$$t_1 = x_1 \cdot t_0$$
$$y = x_0 + t_1$$
$$\textbf{return}\ y$$

$$\Longrightarrow$$

$$P'(x_0,\ x_1, \overline{y}):$$
$$t_0 = \sin(x_0)$$
$$t_1 = x_1 \cdot t_0$$
$$y = x_0 + t_1$$

$$\textbf{return}\ \ \overline{x_0},\ \overline{x_1}$$

- **Goal:** compute $\overline{x_0}$ and $\overline{x_1}$

Adjoint of a variable

$$\overline{v} \equiv \frac{\partial y}{\partial v}$$

The **sensitivity** of the output y to v.

- Adjoints depend on primal values. Add the statements of the original program.
- Compute $\overline{x_1}$ by the chain rule:

$$\overline{x_1} \equiv \frac{\partial y}{\partial x_1} = \frac{\partial y}{\partial t_1}\frac{\partial t_1}{\partial x_1} = \overline{t_1}\frac{\partial t_1}{\partial x_1}$$

Introduction to AD

- **Goal:** compute $\overline{x_0}$ and $\overline{x_1}$

Adjoint of a variable

$$\overline{v} \equiv \frac{\partial y}{\partial v}$$

The **sensitivity** of the output y to v.

$P(x_0,\ x_1):$
$\quad t_0 = \sin(x_0)$
$\quad t_1 = x_1 \cdot t_0$
$\quad y = x_0 + t_1$
$\quad$ **return** y

$\Longrightarrow$

$P'(x_0,\ x_1, \overline{y}):$
$\quad t_0 = \sin(x_0)$
$\quad t_1 = x_1 \cdot t_0$
$\quad y = x_0 + t_1$

$\quad \overline{t_1} = \overline{y}$
$\quad \overline{x_1} = t_0 \cdot \overline{t_1}$

$\quad$ **return** $\overline{x_0},\ \overline{x_1}$

- Adjoints depend on primal values. Add the statements of the original program.
- Compute $\overline{x_1}$ by the chain rule:

$$\overline{x_1} \equiv \frac{\partial y}{\partial x_1} = \frac{\partial y}{\partial t_1}\frac{\partial t_1}{\partial x_1} = \overline{t_1}\frac{\partial t_1}{\partial x_1}$$

Introduction to AD

$$
\begin{array}{l}
P(x_0,\ x_1):\\
\quad t_0 = \sin(x_0)\\
\quad t_1 = x_1 \cdot t_0\\
\quad y = x_0 + t_1\\
\quad \textbf{return } y
\end{array}
\quad \Longrightarrow \quad
\begin{array}{l}
P'(x_0,\ x_1, \overline{y}):\\
\quad t_0 = \sin(x_0)\\
\quad t_1 = x_1 \cdot t_0\\
\quad y = x_0 + t_1\\
\quad \overline{x_0} = \overline{y}\\
\quad \overline{t_1} = \overline{y}\\
\quad \overline{x_1} = t_0 \cdot \overline{t_1}\\
\quad \overline{t_0} = x_1 \cdot \overline{t_1}\\
\quad \overline{x_0} \mathrel{+}= \cos(x_0) \cdot \overline{t_0}\\
\quad \textbf{return } \overline{x_0},\ \overline{x_1}
\end{array}
$$

- **Goal:** compute $\overline{x_0}$ and $\overline{x_1}$

Adjoint of a variable

$$\overline{v} \equiv \frac{\partial y}{\partial v}$$

The **sensitivity** of the output y to v.

- Adjoints depend on primal values. Add the statements of the original program.
- Compute $\overline{x_1}$ by the chain rule.
- Do the same for $\overline{x_0}$.

Introduction to AD

- **Goal:** compute $\overline{x_0}$ and $\overline{x_1}$

$$
\begin{array}{lcl}
P(x_0,\ x_1): & & P'(x_0,\ x_1, \overline{y}): \\
\quad t_0 = \sin(x_0) & & \quad t_0 = \sin(x_0) \\
\quad t_1 = x_1 \cdot t_0 & \Longrightarrow & \quad t_1 = x_1 \cdot t_0 \\
\quad y = x_0 + t_1 & & \quad y = x_0 + t_1 \\
\quad \textbf{return } y & & \quad \overline{x_0} = \overline{y} \\
 & & \quad \overline{t_1} = \overline{y} \\
 & & \quad \overline{x_1} = t_0 \cdot \overline{t_1} \\
 & & \quad \overline{t_0} = x_1 \cdot \overline{t_1} \\
 & & \quad \overline{x_0} \mathrel{+}= \cos(x_0) \cdot \overline{t_0} \\
 & & \quad \textbf{return } \overline{x_0},\ \overline{x_1}
\end{array}
$$

Adjoint of a variable

$$\overline{v} \equiv \frac{\partial y}{\partial v}$$

The **sensitivity** of the output y to v.

- Adjoints depend on primal values. Add the statements of the original program.
- Compute $\overline{x_1}$ by the chain rule.
- Do the same for $\overline{x_0}$.
- Since $\overline{x_0}$ is read twice, its adjoint gets two contributions.

Introduction to AD

$$
\begin{array}{ll}
P(x_0,\ x_1): & P'(x_0,\ x_1, \overline{y}): \\
\quad t_0 = \sin(x_0) & \quad t_0 = \sin(x_0) \\
\quad t_1 = x_1 \cdot t_0 & \quad t_1 = x_1 \cdot t_0 \\
\quad y = x_0 + t_1 & \quad y = x_0 + t_1 \\
\quad \textbf{return}\ y & \quad \overline{x_0} = \overline{y} \\
 & \quad \overline{t_1} = \overline{y} \\
 & \quad \overline{x_1} = t_0 \cdot \overline{t_1} \\
 & \quad \overline{t_0} = x_1 \cdot \overline{t_1} \\
 & \quad \overline{x_0} \mathrel{+}= \cos(x_0) \cdot \overline{t_0} \\
 & \quad \textbf{return}\ \ \overline{x_0},\ \overline{x_1}
\end{array}
$$

$\Longrightarrow$

- **Goal:** compute $\overline{x_0}$ and $\overline{x_1}$

Adjoint of a variable

$$\overline{v} \equiv \frac{\partial y}{\partial v}$$

The **sensitivity** of the output y to v.

- Adjoints depend on primal values. Add the statements of the original program.
- Compute $\overline{x_1}$ by the chain rule.
- Do the same for $\overline{x_0}$.
- Since $\overline{x_0}$ is read twice, its adjoint gets two contributions.
- Adjoints appear in reverse program-order.

Introduction to AD

$$
\begin{array}{l}
P(x_0,\ x_1): \\
\quad t_0 = \sin(x_0) \\
\quad t_1 = x_1 \cdot t_0 \\
\quad y = x_0 + t_1 \\
\quad \textbf{return}\ y
\end{array}
\quad \Longrightarrow \quad
\begin{array}{l}
P'(x_0,\ x_1, \overline{y}): \\
\quad t_0 = \sin(x_0) \\
\quad t_1 = x_1 \cdot t_0 \\
\quad y = x_0 + t_1 \\
\quad \overline{x_0} = \overline{y} \\
\quad \overline{t_1} = \overline{y} \\
\quad \overline{x_1} = t_0 \cdot \overline{t_1} \\
\quad \overline{t_0} = x_1 \cdot \overline{t_1} \\
\quad \overline{x_0} \mathrel{+}= \cos(x_0) \cdot \overline{t_0} \\
\quad \textbf{return}\ \ \overline{x_0},\ \overline{x_1}
\end{array}
$$

- Can express AD as a rewrite rule:

AD rewrite rule

$$
v = f(u, w) \quad \Longrightarrow \quad
\begin{array}{c}
v = f(u, w) \\
\vdots \\
\overline{u} \mathrel{+}= \dfrac{\partial f(u,w)}{\partial u}\overline{v} \\
\overline{w} \mathrel{+}= \dfrac{\partial f(u,w)}{\partial w}\overline{v}
\end{array}
$$

AD Transformation

AD Transformation

- Programs are built from **bodies**; i.e., a list of statements concluding in a result.

$$\left.\begin{array}{l}\left.\begin{array}{l}\left.\mathtt{let}\ x = a + b\right\}\ stm\\ \mathtt{let}\ res = x \ast c\end{array}\right\}\ stms\\ \mathtt{in}\ res\end{array}\right\}\ body$$

AD Transformation

- Programs are built from **bodies**; i.e., a list of statements concluding in a result.

$$\left.\begin{array}{l}\left.\begin{array}{l}\left.\texttt{let}\ x = a + b\right\}\ \mathit{stm}\\ \texttt{let}\ \mathit{res} = x \star c\end{array}\right\}\ \mathit{stms}\\ \texttt{in}\ \mathit{res}\end{array}\right\}\ \mathit{body} \quad\implies\quad \left.\begin{array}{l}\texttt{let}\ x = a+b\\ \texttt{let}\ \mathit{res} = x\star c\end{array}\right\}\ \overrightarrow{\mathit{stms}}$$

- To differentiate:
  1. Execute the statements of the original body; $\overrightarrow{\mathit{stms}}$ is the **forward sweep**.

AD Transformation

- Programs are built from **bodies**; i.e., a list of statements concluding in a result.

$$\left.\begin{array}{l}\left.\begin{array}{l}\left.\mathtt{let}\ x = a + b\right\}\ stm \\ \mathtt{let}\ res = x \ast c\end{array}\right\}\ stms \\ \mathtt{in}\ res\end{array}\right\}\ body \quad\Longrightarrow\quad \left.\begin{array}{l}\left.\begin{array}{l}\mathtt{let}\ x = a + b \\ \mathtt{let}\ res = x \ast c\end{array}\right\}\ \overrightarrow{stms} \\ \left.\begin{array}{l}\mathtt{let}\ \overline{x} = c \ast \overline{res} \\ \mathtt{let}\ \overline{c} \mathrel{+}= x \ast \overline{res} \\ \mathtt{let}\ \overline{a} \mathrel{+}= \overline{x} \\ \mathtt{let}\ \overline{b} \mathrel{+}= \overline{x}\end{array}\right\}\ \overleftarrow{stms}\end{array}\right\}\ \overleftrightarrow{stms}$$

- To differentiate:
  1. Execute the statements of the original body; $\overrightarrow{stms}$ is the **forward sweep**.
  2. Compute the adjoint contributions; $\overleftarrow{stms}$ is the **reverse sweep**.

AD Transformation

- Programs are built from **bodies**; i.e., a list of statements concluding in a result.

$$\left.\begin{array}{l}\left.\begin{array}{l}\left.\mathtt{let}\ x = a + b\right\} stm\\ \mathtt{let}\ res = x * c\end{array}\right\} stms\\ \mathtt{in}\ res\end{array}\right\} body \quad \Longrightarrow \quad \begin{array}{l}\left.\begin{array}{l}\left.\begin{array}{l}\mathtt{let}\ x = a + b\\ \mathtt{let}\ res = x * c\end{array}\right\}\overrightarrow{stms}\\ \left.\begin{array}{l}\mathtt{let}\ \overline{x} = c * \overline{res}\\ \mathtt{let}\ \overline{c} \mathrel{+}= x * \overline{res}\\ \mathtt{let}\ \overline{a} \mathrel{+}= \overline{x}\\ \mathtt{let}\ \overline{b} \mathrel{+}= \overline{x}\end{array}\right\}\overleftarrow{stms}\end{array}\right\}\overleftrightarrow{stms}\\ \left.\color{red}{\mathtt{in}\ (\overline{a},\ \overline{b},\ \overline{c})}\right\}\overline{fvs_{body}}\end{array}$$

- To differentiate:
  1. Execute the statements of the original body; $\overrightarrow{stms}$ is the **forward sweep**.
  2. Compute the adjoint contributions; $\overleftarrow{stms}$ is the **reverse sweep**.
  3. Return the adjoints of **free variables**.

AD by Re-execution

$$\left.\begin{array}{l}
\textbf{let } zs = \textbf{map } (\lambda a\ bs \rightarrow \\
\quad \textbf{let } z = \textbf{reduce } (\lambda x\ y \rightarrow \\
\quad\quad \textbf{let } t = \sin(x) \\
\quad\quad \textbf{let } red_res = t \cdot y \\
\quad\quad \textbf{in } red_res)\ 0\ bs \\
\quad \textbf{let } map_res = z \cdot a \\
\quad \textbf{in } map_res)\ as\ bss \\
\textbf{in } zs
\end{array}\right\}
\begin{array}{l}
stms_0 \\
\quad stms_1 \\
\quad\quad stms_2
\end{array}$$

AD by Re-execution

$$\begin{array}{l} \color{red}{stms_0} \\ \quad \color{blue}{stms_1} \\ \qquad \color{green}{stms_2} \end{array} \implies \begin{array}{l} \overrightarrow{stms_0} \\ \quad \overrightarrow{stms_1} \\ \qquad \overrightarrow{stms_2} \\ \overleftarrow{stms_0} \end{array}$$

AD by Re-execution

$$\begin{array}{l} stms_0 \\ \quad stms_1 \\ \qquad stms_2 \end{array} \Longrightarrow \begin{array}{l} \overrightarrow{stms_0} \\ \quad \overrightarrow{stms_1} \\ \qquad \overrightarrow{stms_2} \\ \overleftarrow{stms_0} \\ \quad \overrightarrow{stms_1} \\ \qquad \overrightarrow{stms_2} \\ \quad \overleftarrow{stms_1} \end{array}$$

AD by Re-execution

$$
\begin{array}{l}
stms_0 \\
\quad stms_1 \\
\qquad stms_2
\end{array}
\implies
\begin{array}{l}
\overrightarrow{stms_0} \\
\quad \overrightarrow{stms_1} \\
\qquad \overrightarrow{stms_2} \\
\overleftarrow{stms_0} \\
\quad \overrightarrow{stms_1} \\
\qquad \overrightarrow{stms_2} \\
\quad \overleftarrow{stms_1} \\
\qquad \overrightarrow{stms_2} \\
\qquad \overleftarrow{stms_2}
\end{array}
$$

- The amount of re-execution is proportional to the depth of the deepest scope.

Re-execution in Perfect Scope Nests

$$\begin{array}{l} stms_0 \\ \quad stms_1 \\ \qquad stms_2 \end{array}$$

General case

$$\begin{array}{l} \texttt{map}\ f \\ \quad \texttt{map}\ g \\ \qquad stms_2 \end{array}$$

Perfect nest

$$\Longrightarrow$$

$$\begin{array}{l} \overrightarrow{stms_0} \\ \quad \overrightarrow{stms_1} \\ \qquad \overrightarrow{stms_2} \\ \overleftarrow{stms_0} \\ \quad \overrightarrow{stms_1} \\ \qquad \overrightarrow{stms_2} \\ \quad \overleftarrow{stms_1} \\ \qquad \overrightarrow{stms_2} \\ \qquad \overleftarrow{stms_2} \end{array}$$

General case

$$\begin{array}{l} \overrightarrow{\texttt{map}\ f} \\ \quad \overrightarrow{\texttt{map}\ g} \\ \qquad \overrightarrow{stms_2} \\ \overleftarrow{\texttt{map}\ f} \\ \quad \overleftarrow{\texttt{map}\ g} \\ \qquad \overrightarrow{stms_2} \\ \qquad \overleftarrow{stms_2} \end{array}$$

Perfect nest

- In perfect scope nests, only the **outermost** and **innermost** scopes are re-executed.

Differentiating Parallel Constructs

`reduce`

- **`reduce`** combines all elements of an array with a binary associative operator $\odot$:

$$\texttt{let}\ y = \texttt{reduce}\ \odot\ e_\odot\ [a_0, a_1, \ldots, a_{n-1}]$$

$$\equiv$$

$$\texttt{let}\ y = a_0 \odot a_1 \odot \cdots \odot a_{n-1}$$

`reduce`

- **`reduce`** combines all elements of an array with a binary associative operator $\odot$:

$$\begin{aligned}\texttt{let}\ y &= \texttt{reduce}\ \odot\ e_{\odot}\ [a_0, a_1, \ldots, a_{n-1}] \\ &\equiv \\ \texttt{let}\ y &= a_0 \odot a_1 \odot \cdots \odot a_{n-1}\end{aligned}$$

- For each a_i in the array, we can group the terms of the reduce as

$$\underbrace{a_0 \odot \cdots \odot a_{i-1}}_{l_i} \odot a_i \odot \underbrace{a_{i+1} \odot \cdots \odot a_{n-1}}_{r_i}$$

`reduce`

- **`reduce`** combines all elements of an array with a binary associative operator $\odot$:

$$\begin{aligned}\texttt{\textbf{let}}\ y &= \texttt{\textbf{reduce}}\ \odot\ e_{\odot}\ [a_0, a_1, \ldots, a_{n-1}]\\ &\equiv \\ \texttt{\textbf{let}}\ y &= a_0 \odot a_1 \odot \cdots \odot a_{n-1}\end{aligned}$$

- For each a_i in the array, we can group the terms of the reduce as

$$\underbrace{a_0 \odot \cdots \odot a_{i-1}}_{l_i} \odot a_i \odot \underbrace{a_{i+1} \odot \cdots \odot a_{n-1}}_{r_i}$$

And then directly apply the AD rewrite rule

$$\overline{a_i}\ \overline{+}{=}\ \frac{\partial(l_i \odot a_i \odot r_i)}{\partial a_i}\,\overline{y}$$

Computing l_i and r_i

- For each $i \in \{0, \ldots, n-1\}$, need to compute l_i and r_i

$$\underbrace{a_0 \odot \cdots \odot a_{i-1}}_{l_i} \odot a_i \odot \underbrace{a_{i+1} \odot \cdots \odot a_{n-1}}_{r_i}$$

Computing l_i and r_i

- For each $i \in \{0, \ldots, n-1\}$, need to compute l_i and r_i

$$\underbrace{a_0 \odot \cdots \odot a_{i-1}}_{l_i} \odot a_i \odot \underbrace{a_{i+1} \odot \cdots \odot a_{n-1}}_{r_i}$$

- For the l_is, do a parallel scan

$$\texttt{let}\ ls = \texttt{scan} \odot e_\odot [a_0, a_1, \ldots, a_{n-1}] \equiv [\underbrace{e_\odot}_{l_0}, \underbrace{a_0}_{l_1}, \underbrace{a_0 \odot a_1}_{l_2}, \ldots, \underbrace{a_0 \odot \ldots \odot a_{n-2}}_{l_{n-1}}]$$

Computing l_i and r_i

- For each $i \in \{0, \ldots, n-1\}$, need to compute l_i and r_i

$$\underbrace{a_0 \odot \cdots \odot a_{i-1}}_{l_i} \odot a_i \odot \underbrace{a_{i+1} \odot \cdots \odot a_{n-1}}_{r_i}$$

- For the l_is, do a parallel scan

$$\mathtt{\textbf{let}}\ ls = \mathtt{\textbf{scan}} \odot e_\odot\, [a_0, a_1, \ldots, a_{n-1}] \equiv [\underbrace{e_\odot}_{l_0}, \underbrace{a_0}_{l_1}, \underbrace{a_0 \odot a_1}_{l_2}, \ldots, \underbrace{a_0 \odot \ldots \odot a_{n-2}}_{l_{n-1}}]$$

- For the rs, the array must be reversed

$$\mathtt{\textbf{let}}\ rs = \mathtt{\textbf{reverse}}\ as \triangleright \mathtt{\textbf{scan}}\ (\lambda x\, y \to y \odot x)\ e_\odot\ [a_0, a_1, \ldots, a_{n-1}] \triangleright \mathtt{\textbf{reverse}}$$

$$\equiv [\underbrace{a_0 \odot \ldots \odot a_{n-2}}_{r_0}, \ldots, \underbrace{a_{n-2} \odot a_{n-1}}_{r_{n-3}}, \underbrace{a_{n-1}}_{r_{n-2}}, \underbrace{e_\odot}_{r_{n-1}}]$$

reduce

- The differentiation of reduce results in the following statements:

$$\left.\textbf{let } y = \textbf{reduce} \odot e_{\odot} [a_0, a_1, \ldots, a_{n-1}]\right\}\text{Forward sweep}$$

$$\vdots$$

$$\left.\begin{aligned}
&\textbf{let } ls = \textbf{scan} \odot e_{\odot}\ as \\
&\textbf{let } rs = \textbf{reverse}\ as \triangleright \textbf{scan}\ (\lambda x\ y \rightarrow y \odot x)\ e_{\odot} \triangleright \textbf{reverse} \\
&\textbf{let } \overline{as} \mp= \textbf{map}\ \left(\lambda l_i\ a_i\ r_i \rightarrow \frac{\partial(l_i \odot a_i \odot r_i)}{\partial a_i}\ \overline{y}\right)\ ls\ as\ rs
\end{aligned}\right\}\text{Reverse sweep}$$

reduce

- The differentiation of reduce results in the following statements:

$$
\left.\begin{array}{l}
\mathbf{let}\ y = \mathbf{reduce}\ \odot\ e_{\odot}\ [a_0, a_1, \ldots, a_{n-1}]
\end{array}\right\}\text{Forward sweep}
$$

$$
\vdots
$$

$$
\left.\begin{array}{l}
\mathbf{let}\ ls = \mathbf{scan}\ \odot\ e_{\odot}\ as \\
\mathbf{let}\ rs = \mathbf{reverse}\ as \triangleright \mathbf{scan}\ (\lambda x\ y \rightarrow y\ \odot\ x)\ e_{\odot} \triangleright \mathbf{reverse} \\
\mathbf{let}\ \overline{as}\ \overline{+}= \mathbf{map}\ \left(\lambda l_i\ a_i\ r_i \rightarrow \frac{\partial(l_i \odot a_i \odot r_i)}{\partial a_i}\ \overline{y}\right)\ ls\ as\ rs
\end{array}\right\}\text{Reverse sweep}
$$

- The rule is **asymptotics-preserving**: scan has the same asymptotics as reduce.

reduce

- The differentiation of reduce results in the following statements:

$$
\left.\textbf{let } y = \textbf{reduce } \odot\ e_\odot\ [a_0, a_1, \ldots, a_{n-1}]\right\}\text{Forward sweep}
$$

$$
\vdots
$$

$$
\left.\begin{aligned}
&\textbf{let } ls = \textbf{scan } \odot\ e_\odot\ as\\
&\textbf{let } rs = \textbf{reverse } as \triangleright \textbf{scan } (\lambda x\, y \to y \odot x)\, e_\odot \triangleright \textbf{reverse}\\
&\textbf{let } \overline{as} \mathrel{\overline{+}=} \textbf{map } \left(\lambda l_i\, a_i\, r_i \to \frac{\partial(l_i \odot a_i \odot r_i)}{\partial a_i}\,\overline{y}\right)\ ls\ as\ rs
\end{aligned}\right\}\text{Reverse sweep}
$$

- The rule is **asymptotics-preserving**: scan has the same asymptotics as reduce.
- **Specialized rules** for other operators ($+$, min, max, $\cdot$) admit even more efficient implementations.

`map`

- Consider the following **`map`** :

$$\texttt{let}\ xs = \texttt{map}\ (\lambda a\ b \to \texttt{let}\ res = a\ \cdot\ b\ \texttt{in}\ res)\ as\ bs$$

map

- Consider the following **map** :

$$\texttt{let}\ xs = \texttt{map}\ (\lambda a\ b \rightarrow \texttt{let}\ res = a \cdot b\ \texttt{in}\ res)\ as\ bs$$

- Differentiating **map** is straightforward: just differentiate the lambda and pass in the necessary adjoints as well:

$$\begin{aligned}\texttt{let}\ \overline{as}, \overline{bs} = \texttt{map}\ (\lambda a\ b\ \overline{x}\ \overline{a_0}\ \overline{b_0} \rightarrow\ & \\ & \texttt{let}\ res = a \cdot b \\ & \texttt{let}\ \overline{a} = b \cdot \overline{x} + \overline{a_0} \\ & \texttt{let}\ \overline{b} = a \cdot \overline{x} + \overline{b_0} \\ & \texttt{in}\ \overline{a}, \overline{b})\ as\ bs\ \overline{xs}\ \overline{as_0}\ \overline{bs_0}\end{aligned}$$

`map` with Free Variables

- **`map`**s involving **free variables** are more complicated to differentiate

$$\texttt{\textbf{let}}\ xs = \texttt{\textbf{map}}\ (\lambda a \rightarrow a \cdot b)\ as$$

`map` with Free Variables

- **`map`**s involving **free variables** are more complicated to differentiate

$$\mathbf{\mathtt{let}}\ xs = \mathbf{\mathtt{map}}\ (\lambda a \to a \cdot b)\ as$$

- **Naive approach:** turn free variables into bound variables.

$$\mathbf{\mathtt{let}}\ xs = \mathbf{\mathtt{map}}\ (\lambda a\ b' \to a \cdot b')\ as\ (\mathbf{\mathtt{replicate}}\ n\ b)$$

`map` with Free Variables

- `map`s involving **free variables** are more complicated to differentiate

$$\texttt{let}\ xs = \texttt{map}\ (\lambda a \to a \cdot b)\ as$$

- **Naive approach:** turn free variables into bound variables.

$$\texttt{let}\ xs = \texttt{map}\ (\lambda a\ b' \to a \cdot b')\ as\ (\texttt{replicate}\ n\ b)$$

- **Problem:** asymptotically **inefficient** for partially used free arrays.

$$\texttt{map}\ (\lambda(i,\ as') \to as'[i])\ is\ (\texttt{replicate}\ n\ as),$$

Efficient `maps` with Free Variables

- In an impure language, asymptotics-preserving adjoint updates for free array variables can be implemented as a **generalized reduction**.

Efficient `map`s with Free Variables

- In an impure language, asymptotics-preserving adjoint updates for free array variables can be implemented as a **generalized reduction**.
  - The adjoint of a free array variable $as[i]$ can be updated with an operation $\overline{as}[i] \mathrel{+}= v$.

Efficient `maps` with Free Variables

- In an impure language, asymptotics-preserving adjoint updates for free array variables can be implemented as a **generalized reduction**.
  - The adjoint of a free array variable $as[i]$ can be updated with an operation $\overline{as}[i] \mathrel{+}= v$.
- In our pure setting, we introduce **accumulators**.
  - **Write-only** view of an array.
  - Preserves purely functional reasoning in the compiler.
  - Preserves asymptotics by operationally doing in-place updates.

Loops

Loops

- **Loops** in Futhark are sugar for tail-recursive functions.
- **Loop parameters** are variables which are variant through the loop and are returned as the result of the loop.

$$
\begin{array}{l}
\textbf{loop}\ y = 2\ \textbf{for}\ i = 0 \ldots n-1\ \textbf{do} \\
\quad \textbf{let}\ y' = y * y \\
\quad \textbf{in}\ y'
\end{array}
$$

$$
\begin{array}{l}
y = 2 \\
\textbf{for}\ i = 0 \ldots n-1\ \textbf{do} \\
\quad y = y * y
\end{array}
$$

(Imperative analog)

Loops

- **Loops** in Futhark are sugar for tail-recursive functions.
- **Loop parameters** are variables which are variant through the loop and are returned as the result of the loop.

$$\begin{array}{l} \texttt{loop}\ y = 2\ \texttt{for}\ i = 0 \ldots n-1\ \texttt{do} \\ \quad \texttt{let}\ y' = y * y \\ \quad \texttt{in}\ y' \end{array}$$

$$\begin{array}{l} y = 2 \\ \texttt{for}\ i = 0 \ldots n-1\ \texttt{do} \\ \quad y = y * y \end{array}$$

(Imperative analog)

- Since the adjoints of the **loop body** are computed in **reverse order**, the loop parameter y needs to be **saved** for each iteration.

Differentiating Loops

$$
\begin{array}{l}
\mathbf{let}\ y'' = \\
\ \mathbf{loop}\ y = y_0\ \mathbf{for}\ i = 0 \ldots n-1\ \mathbf{do} \\
\quad stms_{loop} \\
\quad \mathbf{in}\ y'
\end{array}
$$

1. Execute the original loop, save the value of y in each iteration in ys.

$$
\left.\begin{array}{l}
\mathbf{let}\ ys_0 = \mathbf{scratch}(n, \\
\qquad\qquad\qquad \mathtt{sizeOf}(y_0)) \\
\mathbf{let}\ (y'', ys) = \\
\ \mathbf{loop}\ (y, ys) = (y_0, ys_0) \\
\ \ \mathbf{for}\ i = 0 \ldots n-1\ \mathbf{do} \\
\quad \mathbf{let}\ ys[i] = y \\
\quad stms_{loop} \\
\quad \mathbf{in}\ (y', ys)
\end{array}\right\}\text{Forward sweep}
$$

$$
\left.\begin{array}{l}
\mathbf{let}\ (\overline{y'''}, \overline{fvs_l}) = \\
\ \mathbf{loop}\ (\overline{y}, \overline{fvs_l}) = (\overline{y''}, \overline{fvs_{l_0}}) \\
\ \ \mathbf{for}\ i = n-1 \ldots 0\ \mathbf{do} \\
\quad \mathbf{let}\ y = ys[i] \\
\quad \overrightarrow{stms_{loop}} \\
\quad \overleftarrow{stms_{loop}} \\
\quad \mathbf{in}\ (\overline{y'}, \overline{fvs'_l}) \\
\mathbf{let}\ \overline{y_0} \mathrel{+}= \overline{y'''}
\end{array}\right\}\text{Reverse sweep}
$$

Differentiating Loops

$$
\begin{array}{l}
\mathbf{let}\ y'' = \\
\ \ \mathbf{loop}\ y = y_0\ \mathbf{for}\ i = 0 \ldots n-1\ \mathbf{do} \\
\ \ \ \ stms_{loop} \\
\ \ \ \ \mathbf{in}\ y'
\end{array}
$$

1. Execute the original loop, save the value of y in each iteration in ys.
2. Compute the adjoint contributions of the loop.

$$
\left.
\begin{array}{l}
\mathbf{let}\ ys_0 = \mathbf{scratch}(n, \\
\qquad\qquad\qquad\qquad \mathtt{sizeOf}(y_0)) \\
\mathbf{let}\ (y'', ys) = \\
\ \ \mathbf{loop}\ (y, ys) = (y_0, ys_0) \\
\ \ \ \ \mathbf{for}\ i = 0 \ldots n-1\ \mathbf{do} \\
\ \ \ \ \ \ \mathbf{let}\ ys[i] = y \\
\ \ \ \ \ \ stms_{loop} \\
\ \ \ \ \ \ \mathbf{in}\ (y', ys)
\end{array}
\right\}\text{Forward sweep}
$$

$$
\left.
\begin{array}{l}
\mathbf{let}\ (\overline{y'''}, \overline{fvs_l}) = \\
\ \ \mathbf{loop}\ (\overline{y}, \overline{fvs_l}) = (\overline{y''}, \overline{fvs_{l_0}}) \\
\ \ \ \ \mathbf{for}\ i = n-1 \ldots 0\ \mathbf{do} \\
\ \ \ \ \ \ \mathbf{let}\ y = ys[i] \\
\ \ \ \ \ \ \overrightarrow{stms_{loop}} \\
\ \ \ \ \ \ \overleftarrow{stms_{loop}} \\
\ \ \ \ \ \ \mathbf{in}\ (\overline{y'}, \overline{fvs'_l}) \\
\mathbf{let}\ \overline{y_0} \mathrel{+}= \overline{y'''}
\end{array}
\right\}\text{Reverse sweep}
$$

Differentiating Loops

$$
\begin{array}{l}
\textbf{let}\ y'' = \\
\quad \textbf{loop}\ y = y_0\ \textbf{for}\ i = 0 \ldots n-1\ \textbf{do} \\
\quad\quad \mathit{stms}_{loop} \\
\quad\quad \textbf{in}\ y'
\end{array}
$$

1. Execute the original loop, save the value of *y* in each iteration in *ys*.
2. Compute the adjoint contributions of the loop.
   - Run the loop backwards

$$
\begin{array}{l}
\textbf{let}\ ys_0 = \textbf{scratch}(n, \\
\qquad\qquad\qquad\qquad \texttt{sizeOf}(y_0)) \\
\textbf{let}\ (y'', ys) = \\
\ \textbf{loop}\ (y, ys) = (y_0, ys_0) \\
\ \ \textbf{for}\ i = 0 \ldots n-1\ \textbf{do} \\
\ \ \ \textbf{let}\ ys[i] = y \\
\ \ \ \mathit{stms}_{loop} \\
\ \ \ \textbf{in}\ (y', ys)
\end{array}
$$

Forward sweep

$$
\begin{array}{l}
\textbf{let}\ (\overline{y'''}, \overline{fvs_l}) = \\
\ \textbf{loop}\ (\overline{y}, \overline{fvs_l}) = (\overline{y''}, \overline{fvs_{l_0}}) \\
\ \ \textbf{for}\ i = n-1 \ldots 0\ \textbf{do} \\
\ \ \ \textbf{let}\ y = ys[i] \\
\ \ \ \overrightarrow{\mathit{stms}_{loop}} \\
\ \ \ \overleftarrow{\mathit{stms}_{loop}} \\
\ \ \ \textbf{in}\ (\overline{y'}, \overline{fvs'_l}) \\
\textbf{let}\ \overline{y_0} \mathrel{+}= \overline{y'''}
\end{array}
$$

Reverse sweep

Differentiating Loops

$$
\begin{array}{l}
\textbf{let } y'' = \\
\quad \textbf{loop } y = y_0 \textbf{ for } i = 0 \ldots n-1 \textbf{ do} \\
\quad\quad stms_{loop} \\
\quad\quad \textbf{in } y'
\end{array}
$$

1. Execute the original loop, save the value of y in each iteration in ys.
2. Compute the adjoint contributions of the loop.
   - Run the loop backwards
   - Restore the value of y from ys

$$
\left.
\begin{array}{l}
\textbf{let } ys_0 = \textbf{scratch}(n, \\
\qquad\qquad\qquad\qquad \texttt{sizeOf}(y_0)) \\
\textbf{let } (y'', ys) = \\
\quad \textbf{loop } (y, ys) = (y_0, ys_0) \\
\quad\quad \textbf{for } i = 0 \ldots n-1 \textbf{ do} \\
\quad\quad\quad \textbf{let } ys[i] = y \\
\quad\quad\quad stms_{loop} \\
\quad\quad\quad \textbf{in } (y', ys)
\end{array}
\right\} \text{Forward sweep}
$$

$$
\left.
\begin{array}{l}
\textbf{let } (\overline{y'''}, \overline{fvs_l}) = \\
\quad \textbf{loop } (\overline{y}, \overline{fvs_l}) = (\overline{y''}, \overline{fvs_{l_0}}) \\
\quad\quad \textbf{for } i = n-1 \ldots 0 \textbf{ do} \\
\quad\quad\quad \textbf{let } y = ys[i] \\
\quad\quad\quad \overrightarrow{stms_{loop}} \\
\quad\quad\quad \overleftarrow{stms_{loop}} \\
\quad\quad\quad \textbf{in } (\overline{y'}, \overline{fvs'_l}) \\
\textbf{let } \overline{y_0} \mathrel{+}= \overline{y'''}
\end{array}
\right\} \text{Reverse sweep}
$$

Differentiating Loops

$$
\begin{array}{l}
\mathbf{let}\ y'' = \\
\quad \mathbf{loop}\ y = y_0\ \mathbf{for}\ i = 0 \ldots n-1\ \mathbf{do} \\
\quad\quad \mathit{stms}_{loop} \\
\quad\quad \mathbf{in}\ y'
\end{array}
$$

1. Execute the original loop, save the value of y in each iteration in ys.
2. Compute the adjoint contributions of the loop.
   - Run the loop backwards
   - Restore the value of y from ys
   - Re-execute the body of the original loop

$$
\begin{array}{l}
\mathbf{let}\ ys_0 = \mathbf{scratch}(n, \\
\qquad\qquad\qquad\qquad \mathtt{sizeOf}(y_0)) \\
\mathbf{let}\ (y'', ys) = \\
\quad \mathbf{loop}\ (y, ys) = (y_0, ys_0) \\
\quad\ \ \mathbf{for}\ i = 0 \ldots n-1\ \mathbf{do} \\
\quad\quad \mathbf{let}\ ys[i] = y \\
\quad\quad \mathit{stms}_{loop} \\
\quad\quad \mathbf{in}\ (y', ys)
\end{array}
\Bigg\}\ \text{Forward sweep}
$$

$$
\begin{array}{l}
\mathbf{let}\ (\overline{y'''}, \overline{fvs_l}) = \\
\quad \mathbf{loop}\ (\overline{y}, \overline{fvs_l}) = (\overline{y''}, \overline{fvs_{l_0}}) \\
\quad\ \ \mathbf{for}\ i = n-1 \ldots 0\ \mathbf{do} \\
\quad\quad \mathbf{let}\ y = ys[i] \\
\quad\quad \overrightarrow{\mathit{stms}_{loop}} \\
\quad\quad \overleftarrow{\mathit{stms}_{loop}} \\
\quad\quad \mathbf{in}\ (\overline{y'}, \overline{fvs'_l}) \\
\mathbf{let}\ \overline{y_0} \mathrel{+}= \overline{y'''}
\end{array}
\Bigg\}\ \text{Reverse sweep}
$$

Differentiating Loops

let y'' =
  loop $y = y_0$ **for** $i = 0 \ldots n-1$ **do**
    $stms_{loop}$
    in y'

1. Execute the original loop, save the value of y in each iteration in ys.
2. Compute the adjoint contributions of the loop.
   - Run the loop backwards
   - Restore the value of y from ys
   - Re-execute the body of the original loop
   - Compute the adjoints of the body

let ys_0 = **scratch**(n,
                    `sizeOf`(y_0))
let (y'', ys) =
  loop $(y, ys) = (y_0, ys_0)$
   for $i = 0 \ldots n-1$ **do**
    let $ys[i] = y$
    $stms_{loop}$
    in (y', ys)

Forward sweep

let $(\overline{y'''}, \overline{fvs_l})$ =
  loop $(\overline{y}, \overline{fvs_l}) = (\overline{y''}, \overline{fvs_{l_0}})$
   for $i = n-1 \ldots 0$ **do**
    let $y = ys[i]$
    $\overrightarrow{stms_{loop}}$
    $\overleftarrow{stms_{loop}}$
    in $(\overline{y'}, \overline{fvs'_l})$
let $\overline{y_0}$ += $\overline{y'''}$

Reverse sweep

Loop Strip-mining

- **Loop strip-mining** partitions a loop into a loop nest

$$
\begin{array}{l}
\textbf{loop}\ y = y_0\ \textbf{for}\ i = 0 \ldots n^3 - 1\ \textbf{do} \\
\quad \mathit{stms}
\end{array}
\implies
\begin{array}{l}
\textbf{loop}\ y_j = y_0\ \textbf{for}\ j = 0 \ldots n - 1\ \textbf{do} \\
\quad \textbf{loop}\ y_k = y_j\ \textbf{for}\ k = 0 \ldots n - 1\ \textbf{do} \\
\quad\quad \textbf{loop}\ y_m = y_k\ \textbf{for}\ m = 0 \ldots n - 1\ \textbf{do} \\
\quad\quad\quad \textbf{let}\ i = j * n^{2/3} + k * n^{1/3} + m \\
\quad\quad\quad \mathit{stms}
\end{array}
$$

Loop Strip-mining

- **Loop strip-mining** partitions a loop into a loop nest

$$
\begin{array}{l}
\mathbf{\mathtt{loop}}\ y = y_0\ \mathbf{\mathtt{for}}\ i = 0 \ldots n^3 - 1\ \mathbf{do} \\
\quad stms
\end{array}
\quad \Longrightarrow \quad
\begin{array}{l}
\mathbf{\mathtt{loop}}\ y_j = y_0\ \mathbf{\mathtt{for}}\ j = 0 \ldots n - 1\ \mathbf{do} \\
\quad \mathbf{\mathtt{loop}}\ y_k = y_j\ \mathbf{\mathtt{for}}\ k = 0 \ldots n - 1\ \mathbf{do} \\
\quad\quad \mathbf{\mathtt{loop}}\ y_m = y_k\ \mathbf{\mathtt{for}}\ m = 0 \ldots n - 1\ \mathbf{do} \\
\quad\quad\quad \mathbf{\mathtt{let}}\ i = j * n^{2/3} + k * n^{1/3} + m \\
\quad\quad\quad stms
\end{array}
$$

- For the original loop, we save n^3 versions of y on the tape.

Loop Strip-mining

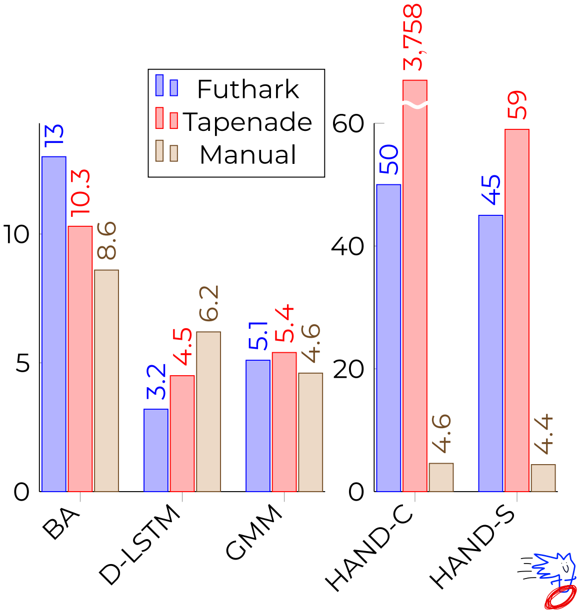
- **Loop strip-mining** partitions a loop into a loop nest

$$\begin{array}{l}\texttt{loop}\ y = y_0\ \texttt{for}\ i = 0 \ldots n^3 - 1\ \texttt{do} \\ \quad stms\end{array} \implies \begin{array}{l}\texttt{loop}\ y_j = y_0\ \texttt{for}\ j = 0 \ldots n - 1\ \texttt{do} \\ \quad \texttt{loop}\ y_k = y_j\ \texttt{for}\ k = 0 \ldots n - 1\ \texttt{do} \\ \quad\quad \texttt{loop}\ y_m = y_k\ \texttt{for}\ m = 0 \ldots n - 1\ \texttt{do} \\ \quad\quad\quad \texttt{let}\ i = j * n^{2/3} + k * n^{1/3} + m \\ \quad\quad\quad stms\end{array}$$

- For the original loop, we save n^3 versions of y on the tape.
- For the strip-mined loop, only $3n$ versions are saved. (With an increased re-execution overhead factor of 3.)

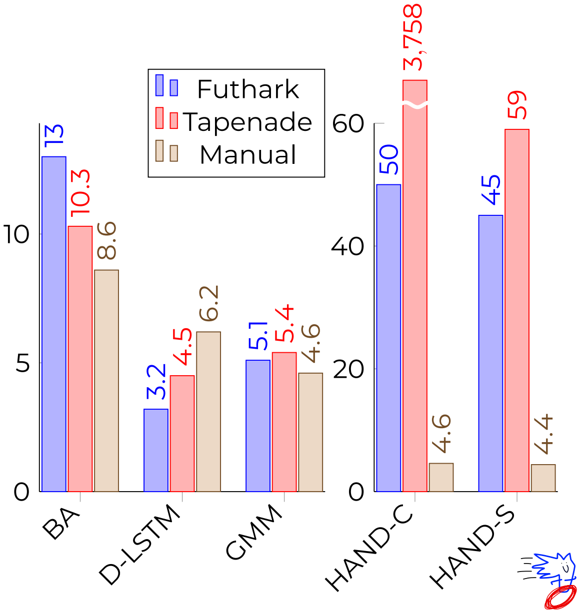
Benchmarks

CPU Benchmarks - ADBench



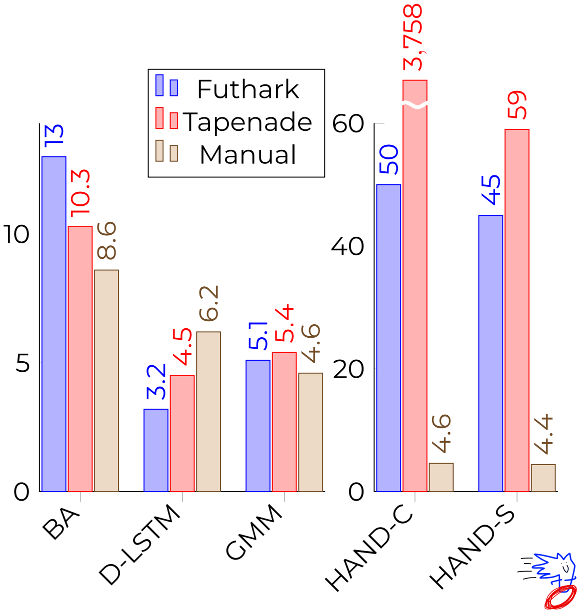
- ADBench: a collection of AD benchmarks for comparing sequential AD tools.

CPU Benchmarks - ADBench



- ADBench: a collection of AD benchmarks for comparing sequential AD tools.
- Benchmarked Futhark using its C backend.

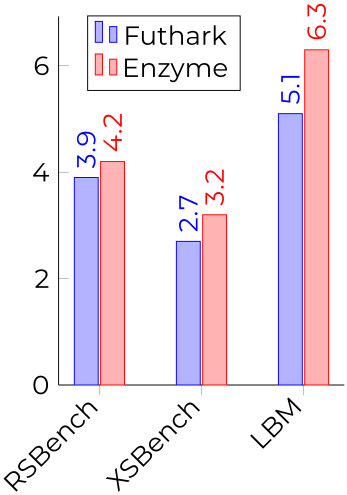
CPU Benchmarks - ADBench



- ADBench: a collection of AD benchmarks for comparing sequential AD tools.
- Benchmarked Futhark using its C backend.
- Performance measured in **AD overhead**:

$$\frac{\text{differentiated runtime}}{\text{original runtime}}$$

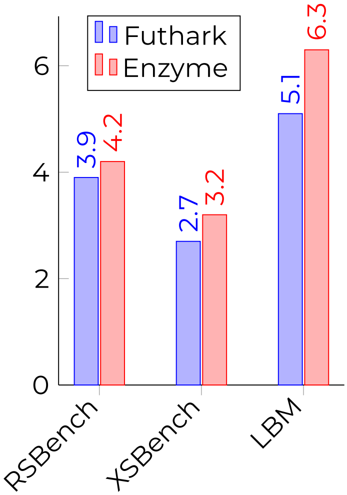
GPU Benchmarks - vs. Enzyme



- Performance measured in **AD overhead**:

$$\frac{\text{differentiated runtime}}{\text{original runtime}}$$

GPU Benchmarks - vs. Enzyme

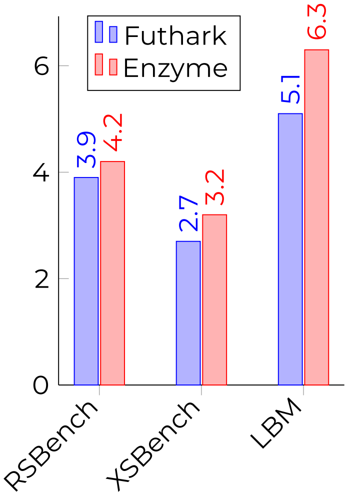


- Performance measured in **AD overhead**:

$$\frac{\text{differentiated runtime}}{\text{original runtime}}$$

- Enzyme is LLVM compiler plugin that performs AD on a low-level imperative IR.

GPU Benchmarks - vs. Enzyme

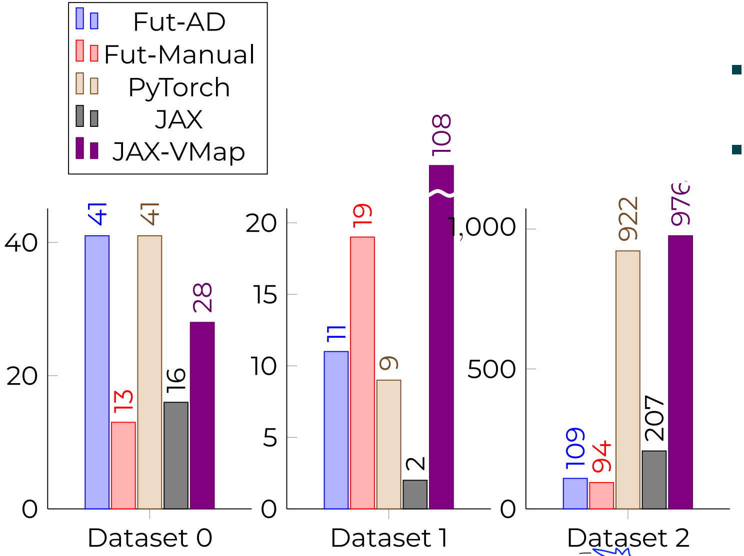


- Performance measured in **AD overhead**:

$$\frac{\text{differentiated runtime}}{\text{original runtime}}$$

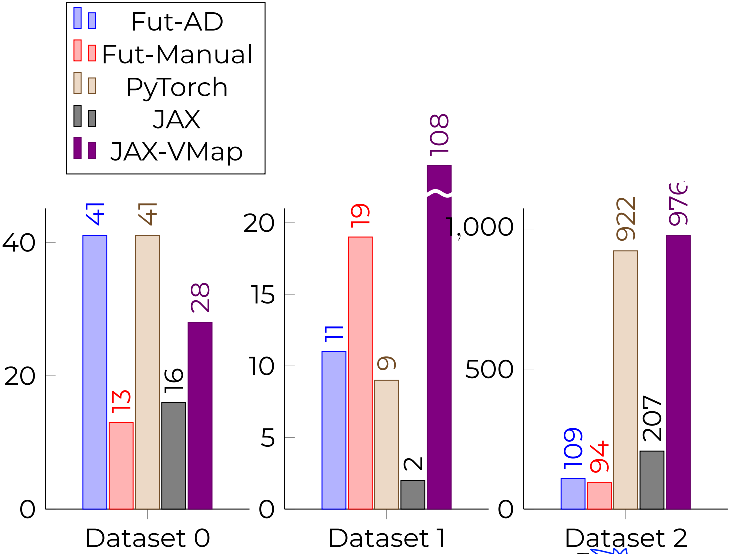
- Enzyme is LLVM compiler plugin that performs AD on a low-level imperative IR.
- RSBench and XSBench are comprised of a large parallell loop with inner sequential loops and branches.
- LBM consists of a large sequential loop containing a parallel loop.

GPU Benchmarks - *k*-means



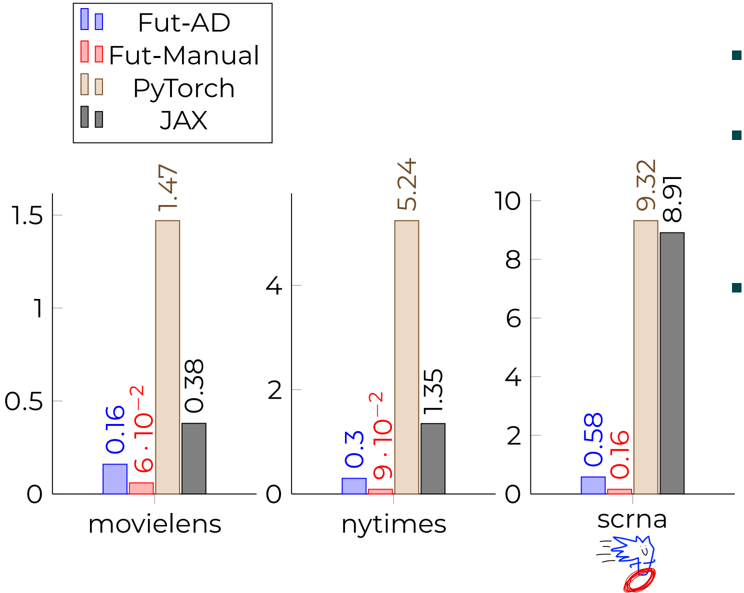
- Performance measured in **miliseconds**.
- *k*-means clustering using AD-based Newton's method to find cluster centers.

GPU Benchmarks - *k*-means



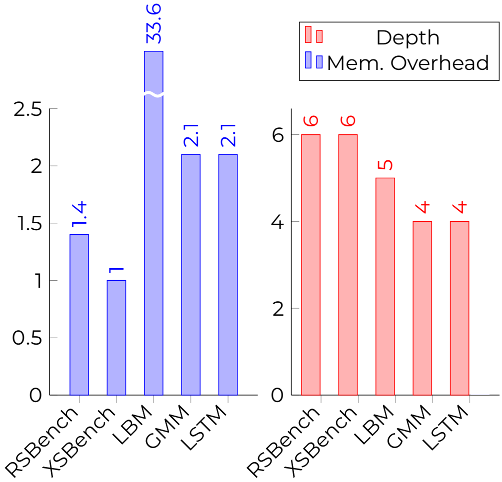
- Performance measured in **miliseconds**.
- *k*-means clustering using AD-based Newton's method to find cluster centers.
- PyTorch and JAX use hand-tuned matrix primitives; JAX(vmap) instead uses JAX's vectorizing map operation for these operations, in analog with Futhark.

GPU Benchmarks - Sparse k-means



- Performance measured in **seconds**.
- PyTorch and JAX use hand-tuned matrix primitives and sparse libraries.
- Futhark just uses a standard CSR implementation.

GPU Benchmarks - Depth and Memory Consumption



- **AD Memory overhead**:

$$\frac{\text{differentiated mem. consumption}}{\text{original mem. consumption}}$$

- With **loop strip-mining**, LBM's memory overhead is **reduced to 8.7**, with only a 1.3× increase in runtime.
- **Strong performance** on programs with **non-trivial depth** demonstrates the viability of a recomputation-based approach to AD.

Conclusions

Conclusions

- AD in a **nested-parallel**, **high-level** and **hardware-neutral** functional language.

Conclusions

- AD in a **nested-parallel**, **high-level** and **hardware-neutral** functional language.
- **Key idea:** high-level differentiation using specialized rules for parallel combinators.

Conclusions

- AD in a **nested-parallel**, **high-level** and **hardware-neutral** functional language.
- **Key idea:** high-level differentiation using specialized rules for parallel combinators.
- **Key idea:** re-computation instead of a tape (except for loops!).

Conclusions

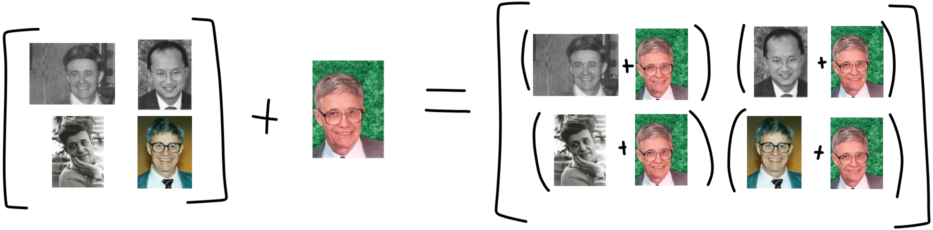
- AD in a **nested-parallel**, **high-level** and **hardware-neutral** functional language.
- **Key idea:** high-level differentiation using specialized rules for parallel combinators.
- **Key idea:** re-computation instead of a tape (except for loops!).
- Strong performance against state-of-the-art AD competitors.

Conclusions

- AD in a **nested-parallel**, **high-level** and **hardware-neutral** functional language.
- **Key idea:** high-level differentiation using specialized rules for parallel combinators.
- **Key idea:** re-computation instead of a tape (except for loops!).
- Strong performance against state-of-the-art AD competitors.
- The implementation is now mature and available in the Futhark compiler.

Thing #2: **AUTOMAP**

Robert Schenck, Nikolaj Hey Hinnerskov, Troels Henriksen,
Magnus Madsen, Martin Elsman



Rank polymorphism

- `1 + 2 => 3`

Rank polymorphism

- `1 + 2 => 3`
- `[1,2,3] + [4,5,6] => [5,7,9]`

Rank polymorphism

- `1 + 2 => 3`
- `[1,2,3] + [4,5,6] => [5,7,9]`
- `[1,2,3] + 4 => [5,6,7]`

Rank polymorphism

- `1 + 2 => 3`
- `[1,2,3] + [4,5,6] => [5,7,9]`
- `[1,2,3] + 4 => [5,6,7]`
- `sqrt [[1,4,9], [16,25,36]] => [[1,2,3], [4,5,6]]`

Rank polymorphism

Rank polymorphism

The ability to apply functions to arguments with different ranks than the function expects.

Rank polymorphism

Rank polymorphism

The ability to apply functions to arguments with different ranks than the function expects.

- Makes code easier to read, more enjoyable to write, and closer to math:

`map (+) [1,2,3] [4,5,6]` vs. `[1,2,3] + [4,5,6]`

Rank polymorphism

Rank polymorphism

The ability to apply functions to arguments with different ranks than the function expects.

- Makes code easier to read, more enjoyable to write, and closer to math:

  `map (+) [1,2,3] [4,5,6]` vs. `[1,2,3] + [4,5,6]`

- This work: how do we get **rank polymorphic applications** in a statically-typed language with parametric polymorphism?

`map` and `rep`

- `map f xs` applies `f` to each element of `xs`:

```
map f [x_0, x_1, ..., x_n] = [f x_0, f x_1, ..., f x_n]
```

`map` and `rep`

- `map f xs` applies `f` to each element of `xs`:

```
map f [x_0, x_1, ..., x_n] = [f x_0, f x_1, ..., f x_n]
```

- You can **`map`** functions that take multiple arguments too:

```
map (+) [x_0, ..., x_n] [y_0, ..., y_n]
 = [x_0 + y_0, ..., x_n + y_n]
```

`map` and `rep`

- `map f xs` applies `f` to each element of `xs`:

  ```
  map f [x_0, x_1, ..., x_n] = [f x_0, f x_1, ..., f x_n]
  ```

- You can `map` functions that take multiple arguments too:

  ```
  map (+) [x_0, ..., x_n] [y_0, ..., y_n]
   = [x_0 + y_0, ..., x_n + y_n]
  ```

- `rep x` makes an array of unspecified length whose elements are all `x`:

  ```
  rep x = [x, x, ..., x]
  ```

  - We'll ignore the question of how many elements are needed.

An example

```
[[1,2],[3,4]] + 1
```

An example

```
[[1,2],[3,4]] + 1
```

elaborates to

```
[[1,2],[3,4]] + rep (rep 1)
```

An example

```
[[1,2],[3,4]] + 1
```

elaborates to

```
[[1,2],[3,4]] + rep (rep 1)
```

which further elaborates to

```
map (map (+)) [[1,2],[3,4]]
     (rep (rep 1))
```

An example

```
[[1,2],[3,4]] + 1
```

elaborates to

```
[[1,2],[3,4]] + rep (rep 1)
```

which further elaborates to

```
map (map (+)) [[1,2],[3,4]]
     (rep (rep 1))
```

```
xss : [][]int
yss : [][]int
f: []int → [][]int → int
--------------------------
f xss yss
```

An example

```
[[1,2],[3,4]] + 1
```

elaborates to

```
[[1,2],[3,4]] + rep (rep 1)
```

which further elaborates to

```
map (map (+)) [[1,2],[3,4]]
      (rep (rep 1))
```

```
xss : [][]int
yss : [][]int
f: []int → [][]int → int
-------------------------------
f xss yss
```

First, we map `f` across both matrices:

```
map f xss yss
```

An example

```
[[1,2],[3,4]] + 1
```

elaborates to

```
[[1,2],[3,4]] + rep (rep 1)
```

which further elaborates to

```
map (map (+)) [[1,2],[3,4]]
      (rep (rep 1))
```

```
xss : [][]int
yss : [][]int
f: []int → [][]int → int
-------------------------
f xss yss
```

First, we map `f` across both matrices:

```
map f xss yss
```

Because of the **map**, `yss` must be replicated:

```
map f xss (rep yss)
```

An example

```
[[1,2],[3,4]] + 1
```

elaborates to

```
[[1,2],[3,4]] + rep (rep 1)
```

which further elaborates to

```
map (map (+)) [[1,2],[3,4]]
        (rep (rep 1))
```

```
xss : [][]int
yss : [][]int
f: []int → [][]int → int
-------------------------
f xss yss
```

First, we map `f` across both matrices:

```
map f xss yss
```

Because of the **map**, `yss` must be replicated:

```
map f xss (rep yss)
```

`rep`s can often be eliminated

```
map (λxs → f xs yss) xss
```

Goal

For each function application, the compiler should automatically insert **`map`**s or **`rep`**s to make the application **rank-correct**.

Goal

For each function application, the compiler should automatically insert **`map`**s or **`rep`**s to make the application **rank-correct**.

```
f x  ⟹  map ( ... (map f) ... ) (rep ... (rep x) ... )
```

Challenge: ambiguity

```
sum : []int → int
length : []a → int
xss : [][]int
---------------------
sum (length xss)
```

Challenge: ambiguity

```
sum : []int → int
length : []a → int
xss : [][]int
-------------------------
sum (length xss)
```

Many rank-correct elaborations:

1. `sum (rep (length xss))`

Challenge: ambiguity

```
sum : []int → int
length : []a → int
xss : [][]int
---------------------
sum (length xss)
```

Many rank-correct elaborations:

1. `sum (rep (length xss))`
2. `sum (map length xss)`

Challenge: ambiguity

```
sum : []int → int
length : []a → int
xss : [][]int
-----------------------
sum (length xss)
```

Many rank-correct elaborations:

1. `sum (rep (length xss))`
2. `sum (map length xss)`
3. `map sum (map (map length) (rep xss))`
4. ...

The Strategy

Rule 1

An application can be `map`ped or `rep`ped (or neither) but **never both**.

The Strategy

Rule 1

An application can be **`map`**ped or **`rep`**ped (or neither) but **never both**.

- **OK**:
  - `map f x`

The Strategy

Rule 1

An application can be **`map`**ped or **`rep`**ped (or neither) but **never both**.

- **OK**:
  - `map f x`
  - `g (rep (rep x))`

The Strategy

Rule 1

An application can be **`map`**ped or **`rep`**ped (or neither) but **never both**.

- **OK**:
  - `map f x`
  - `g (rep (rep x))`
  - `(map (map h) x) (rep y)`

The Strategy

Rule 1

An application can be **`map`**ped or **`rep`**ped (or neither) but **never both**.

- **OK**:
  - `map f x`
  - `g (rep (rep x))`
  - `(map (map h) x) (rep y)`

  BAD:
  - `map f (rep x)`
  - `(map (map g)) (rep x)`

The Strategy

Rule 1

An application can be **`map`**ped or **`rep`**ped (or neither) but **never both**.

- **OK**:
  - `map f x`
  - `g (rep (rep x))`
  - `(map (map h) x) (rep y)`

  BAD:
  - `map f (rep x)`
  - `(map (map g)) (rep x)`

- Never necessary to **`map`** and **`rep`** in the same application to obtain a rank-correct program.

The Strategy

Rule 2

Minimize the number of inserted **`map`**s and **`rep`**s.

The Strategy

Rule 2

Minimize the number of inserted **`map`**s and **`rep`**s.

- Generally aligns with programmer's intent; makes for a simple mental model.

The Strategy

Rule 2

Minimize the number of inserted **`maps`** and **`reps`**.

- Generally aligns with programmer's intent; makes for a simple mental model.

- The minimization is over **all** the applications of a top-level definition.
  - Only have to choose from the set of minimal solutions.

The Strategy

Rule 2

Minimize the number of inserted **`map`**s and **`rep`**s.

- Generally aligns with programmer's intent; makes for a simple mental model.

- The minimization is over **all** the applications of a top-level definition.
  - Only have to choose from the set of minimal solutions.

    `sum (length xss)` can be elaborated to:

    1. `sum (rep (length xss))`

The Strategy

Rule 2

Minimize the number of inserted **`maps`** and **`reps`**.

- Generally aligns with programmer's intent; makes for a simple mental model.

- The minimization is over **all** the applications of a top-level definition.
  - Only have to choose from the set of minimal solutions.

  `sum (length xss)` can be elaborated to:

  1. `sum (rep (length xss))`
  2. `sum (map length xss)`

The Strategy

Rule 2

Minimize the number of inserted **`map`**s and **`rep`**s.

- Generally aligns with programmer's intent; makes for a simple mental model.

- The minimization is over **all** the applications of a top-level definition.
  - Only have to choose from the set of minimal solutions.

    `sum (length xss)` can be elaborated to:

    1. `sum (rep (length xss))`
    2. `sum (map length xss)`
    3. `map sum (map (map length) (rep xss))`

The Strategy

Rule 2

Minimize the number of inserted **`map`**s and **`rep`**s.

- Generally aligns with programmer's intent; makes for a simple mental model.

- The minimization is over **all** the applications of a top-level definition.
  - Only have to choose from the set of minimal solutions.

    `sum (length xss)` can be elaborated to:

    1. `sum (`**`rep`**` (length xss))`
    2. `sum (`**`map`**` length xss)`
    3. **`map`**` sum (`**`map`**` (`**`map`**` length) (`**`rep`**` xss))`
    4. ...

The Strategy

Rule 2

Minimize the number of inserted **`map`**s and **`rep`**s.

- Generally aligns with programmer's intent; makes for a simple mental model.

- The minimization is over **all** the applications of a top-level definition.
  - Only have to choose from the set of minimal solutions.

    `sum (length xss)` can be elaborated to:

    1. `sum (rep (length xss))`
    2. `sum (map length xss)`
    3. ~~`map sum (map (map length) (rep xss))`~~
    4. ~~...~~

Challenge: elaboration is global

- `sum (length xss)` can be elaborated to:
  1. `sum (rep (length xss))`
     - **Local** reasoning: in the application of `sum` to `length xss`, the argument is underdimensioned, so a **`rep`** is inserted.

Challenge: elaboration is global

- `sum (length xss)` can be elaborated to:
  1. `sum (rep (length xss))`
     - **Local** reasoning: in the application of `sum` to `length xss`, the argument is underdimensioned, so a **`rep`** is inserted.
  2. `sum (map length xss)`
     - **Global** reasoning: `length xss` is rank-correct as-is, but a **`map`** is inserted because of the outer `sum` application.

Challenge: elaboration is global

- `sum (length xss)` can be elaborated to:
  1. `sum (rep (length xss))`
     - **Local** reasoning: in the application of `sum` to `length xss`, the argument is underdimensioned, so a **`rep`** is inserted.
  2. `sum (map length xss)`
     - **Global** reasoning: `length xss` is rank-correct as-is, but a **`map`** is inserted because of the outer `sum` application.

- Elaborations of inner applications affect outer applications.
  - To find all minimal elaborations, must consider all applications **simultaneously**.

Challenge: type variables

- Futhark has parametric polymorphism:

```
id : a → a
length : []a → int
```

Challenge: type variables

- Futhark has parametric polymorphism:

```
id : a → a
length : []a → int
```

- A type variable can have any rank!

Challenge: type variables

- Futhark has parametric polymorphism:

```
id : a → a
length : []a → int
```

- A type variable can have any rank!

- How do we statically insert **`map`**s and **`rep`**s in the presence of type variables, whose ranks aren't known?

Constraints

- Suppose

```
f : p → b
x : a
```

Constraints

- Suppose

```
f : p → b
x : a
```

- The application `f x` has constraint

$$p = a$$

Constraints

- Suppose

$$\begin{aligned}\texttt{f} &: \texttt{p} \to \texttt{b}\\ \texttt{x} &: \texttt{a}\end{aligned}$$

- The application `f x` has constraint

$$p = a$$

- We only care about rank, so relax to

$$|p| = |a|$$

  where $|p|$ is the **rank** of p.
  For example, $|\mathbf{[\,][\,]}\texttt{int}| = 2$ and $|\texttt{int}| = 0$.

Constraints - `map`

Rank polymorphisms means rank differences are allowed.

Constraints - `map`

Rank polymorphisms means rank differences are allowed.

- Case $|p| < |a|$:
  - Introduce a **rank variable** M to account for the difference:

$$M + |p| = |a|$$

Constraints - `map`

Rank polymorphisms means rank differences are allowed.

- Case $|p| < |a|$:
  - Introduce a **rank variable** M to account for the difference:

$$M + |p| = |a|$$

  - `sqrt : int → int`
    `[1,2,3] : []int`

    Application `sqrt [1,2,3]` gives the constraint

$$M + \underbrace{|\texttt{int}|}_{0} = \underbrace{|\texttt{[]int}|}_{1} \quad \Longrightarrow \quad M = 1$$

Constraints - `map`

Rank polymorphisms means rank differences are allowed.

- Case $|p| < |a|$:
  - Introduce a **rank variable** M to account for the difference:

$$M + |p| = |a|$$

  - `sqrt : int → int`
    `[1,2,3] : []int`

    Application `sqrt [1,2,3]` gives the constraint

$$M + \underbrace{|\texttt{int}|}_{0} = \underbrace{|\texttt{[]int}|}_{1} \quad \Longrightarrow \quad M = 1$$

  - M is equal to the number of `map`s required:

```
map sqrt [1,2,3]
```

Constraints - `rep`

- Case $|p| > |a|$:
  - Introduce a rank variable $\color{red}{R}$ to account for the difference:

$$|p| = \color{red}{R} + |a|$$

Constraints - `rep`

- Case $|p| > |a|$:
  - Introduce a rank variable R to account for the difference:

$$|p| = R + |a|$$

  - Example: `length : []b` $\to$ `int` The application `length 3` gives the constraint

$$\begin{aligned} |\mathbf{[\,]}\, b| &= R + |\texttt{int}| \\ 1 + |b| = R \quad &\Longrightarrow \quad R = 1, |b| = 0 \end{aligned}$$

Constraints - `rep`

- Case $|p| > |a|$:
  - Introduce a rank variable R to account for the difference:

$$|p| = R + |a|$$

  - Example: `length : []b → int` The application `length 3` gives the constraint

$$
\begin{aligned}
|\mathbf{[\,]}\, b| &= R + |\texttt{int}| \\
1 + |b| &= R \quad \Longrightarrow \quad R = 1, |b| = 0 \\
& \qquad\qquad\qquad\; R = 2, |b| = 1 \\
& \qquad\qquad\qquad\qquad \ldots
\end{aligned}
$$

Constraints - `rep`

- Case $|p| > |a|$:
  - Introduce a rank variable R to account for the difference:

$$|p| = R + |a|$$

  - Example: `length : []b → int` The application `length 3` gives the constraint

$$\begin{aligned} |[\,]\,b| &= R + |\texttt{int}| \\ 1 + |b| &= R \quad \Longrightarrow \quad R = 1, |b| = 0 \\ & \qquad\qquad\qquad\;\; R = 2, |b| = 1 \\ & \qquad\qquad\qquad\qquad \ldots \end{aligned}$$

  - R is equal to the number of **`rep`**s required:
    - `length (rep 3)`
    - `length (rep (rep 3))`
    - ...

Constraints - Summary

- Each application of a function `f : p → c` to an argument `x : a` generates a constraint

$$M + |p| = R + |a|$$

Constraints - Summary

- Each application of a function `f : p` $\rightarrow$ `c` to an argument `x : a` generates a constraint

$$M + |p| = R + |a|$$

- Rule 1: can either **`map`** or **`rep`** but not both

$$M = 0 \text{ or } R = 0$$

Constraints to ILPs

- Collect the constraints for each function application.

Constraints to ILPs

- Collect the constraints for each function application.
- Example: `sum (length xss)`

Constraints to ILPs

- Collect the constraints for each function application.
- Example: `sum (length xss)`

$$\left.\begin{array}{l} M_1 + 1 + |a| = R_1 + 2 \\ M_1 = 0 \text{ or } R_1 = 0 \end{array}\right\} \texttt{length}$$

Constraints to ILPs

- Collect the constraints for each function application.
- Example: `sum (length xss)`

$$\left.\begin{aligned} &M_1 + 1 + |a| = R_1 + 2 \\ &M_1 = 0 \text{ or } R_1 = 0 \end{aligned}\right\} \texttt{length}$$

$$\left.\begin{aligned} &M_2 + 1 = R_2 + M_1 \\ &M_2 = 0 \text{ or } R_2 = 0 \end{aligned}\right\} \texttt{sum}$$

Constraints to ILPs

- Collect the constraints for each function application.
- Example: `sum (length xss)`

$$\left.\begin{aligned} M_1 + 1 + |a| &= R_1 + 2 \\ M_1 = 0 \text{ or } R_1 &= 0 \end{aligned}\right\} \texttt{length}$$

$$\left.\begin{aligned} M_2 + 1 &= R_2 + M_1 \\ M_2 = 0 \text{ or } R_2 &= 0 \end{aligned}\right\} \texttt{sum}$$

- Rule 2: Minimize the number of **`map`**s and **`rep`**s

Constraints to ILPs

- Collect the constraints for each function application.
- Example: `sum (length xss)`

minimize

$$M_1 + R_1 + M_2 + R_2$$

subject to

$$\left.\begin{array}{l} M_1 + 1 + |a| = R_1 + 2 \\ M_1 = 0 \text{ or } R_1 = 0 \end{array}\right\} \texttt{length}$$

$$\left.\begin{array}{l} M_2 + 1 = R_2 + M_1 \\ M_2 = 0 \text{ or } R_2 = 0 \end{array}\right\} \texttt{sum}$$

- Rule 2: Minimize the number of **`map`**s and **`rep`**s

Constraints to ILPs

- Collect the constraints for each function application.
- Example: `sum (length xss)`

minimize

$$M_1 + R_1 + M_2 + R_2$$

subject to

$$\left.\begin{aligned} M_1 + 1 + |a| &= R_1 + 2 \\ M_1 = 0 \text{ or } R_1 &= 0 \end{aligned}\right\} \texttt{length}$$

$$\left.\begin{aligned} M_2 + 1 &= R_2 + M_1 \\ M_2 = 0 \text{ or } R_2 &= 0 \end{aligned}\right\} \texttt{sum}$$

- Rule 2: Minimize the number of **`map`**s and **`rep`**s
- The or-constraints can be linearized to obtain an integer linear program (ILP).

Automap TL;DR

1. For each application generate rank equality and Rule 1 (**`map`** or **`rep`** but not both) constraints.

AUTOMAP TL;DR

1. For each application generate rank equality and Rule 1 (**`map`** or **`rep`** but not both) constraints.

2. Transform the constraint set into an ILP and solve.

AUTOMAP TL;DR

1. For each application generate rank equality and Rule 1 (**`map`** or **`rep`** but not both) constraints.

2. Transform the constraint set into an ILP and solve.

3. Use the ILP solution to elaborate. E.g., if the *i*-th application `f x` has $M_i = 3$ and $R_i = 0$:

$$\texttt{f x} \quad \Longrightarrow \quad \textbf{map}\ (\textbf{map}\ (\textbf{map}\ \texttt{f}))\ \texttt{x}$$

Automap TL;DR

1. For each application generate rank equality and Rule 1 (**`map`** or **`rep`** but not both) constraints.

2. Transform the constraint set into an ILP and solve.

3. Use the ILP solution to elaborate. E.g., if the *i*-th application `f x` has $M_i = 3$ and $R_i = 0$:

$$\texttt{f x} \implies \textbf{map} \; (\textbf{map} \; (\textbf{map} \; \texttt{f})) \; \texttt{x}$$

4. Type check elaborated program and continue with compilation as usual.

User experience

- **`map`** and **`rep`** are normal source-level functions.
  - Programmer free to use AUTOMAP to whatever extent they wish.

User experience

- **`map`** and **`rep`** are normal source-level functions.
  - Programmer free to use AUTOMAP to whatever extent they wish.

Ambiguity feedback:

```
Error: sum (length xss) has multiple elaborations:
  1. sum (rep (length xss))
  2. sum (map length xss)
```

  - Nice error messages.
  - Disambiguation is easy: just insert a **`map`** or **`rep`** into the source.

User experience

- **`map`** and **`rep`** are normal source-level functions.
  - Programmer free to use AUTOMAP to whatever extent they wish.

Ambiguity feedback:

```
Error: sum (length xss) has multiple elaborations:
  1. sum (rep (length xss))
  2. sum (map length xss)
```

  - Nice error messages.
  - Disambiguation is easy: just insert a **`map`** or **`rep`** into the source.

- Fully transparent: the compiler can always elaborate any implicit **`map`**s or **`rep`**s.

User experience

- **`map`** and **`rep`** are normal source-level functions.
  - Programmer free to use AUTOMAP to whatever extent they wish.

Ambiguity feedback:

```
Error: sum (length xss) has multiple elaborations:
  1. sum (rep (length xss))
  2. sum (map length xss)
```

  - Nice error messages.
  - Disambiguation is easy: just insert a **`map`** or **`rep`** into the source.

- Fully transparent: the compiler can always elaborate any implicit **`map`**s or **`rep`**s.

Practical impact

- We implemented AUTOMAP in **Futhark**, a functional array language that supports parametric polymorphism and top-level let-polymorphism.

Practical impact

- We implemented AutoMap in **Futhark**, a functional array language that supports parametric polymorphism and top-level let-polymorphism.

- Difficult to quantify value of feature that is glorified syntax sugar.

Practical impact

- We implemented Automap in **Futhark**, a functional array language that supports parametric polymorphism and top-level let-polymorphism.
- Difficult to quantify value of feature that is glorified syntax sugar.
- We (manually!) rewrote programs to take advantage of Automap when we judged it improved readability.

Practical impact: before

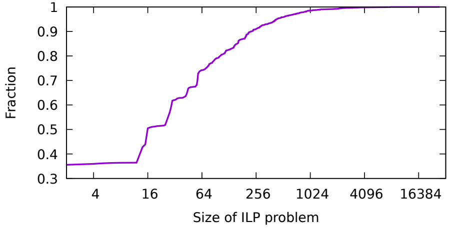
```
def main [nK][nX]
         (kx: [nK]f32) (ky: [nK]f32) (kz: [nK]f32)
         (x: [nX]f32) (y: [nX]f32) (z: [nX]f32)
         (phiR: [nK]f32) (phiI: [nK]f32)
       : ([nX]f32, [nX]f32) =
  let phiM = map (λr i → r*r + i*i) phiR phiI
  let as = map (λx_e y_e z_e →
                 map (2*pi*)
                   (map (λkx_e ky_e kz_e →
                      kx_e*x_e + ky_e*y_e + kz_e*z_e)
                     kx ky kz))
               x y z
  let qr = map (λa → sum(map2 (*) phiM (map cos a))) as
  let qi = map (λa → sum(map2 (*) phiM (map sin a))) as
  in (qr, qi)
```

Practical impact: after

```
def main [nK][nX]
         (kx: [nK]f32) (ky: [nK]f32) (kz: [nK]f32)
         (x: [nX]f32) (y: [nX]f32) (z: [nX]f32)
         (phiR: [nK]f32) (phiI: [nK]f32)
       : ([nX]f32, [nX]f32) =
  let phiM = phiR*phiR + phiI*phiI
  let as = 2*pi*(kx*transpose (rep x)
               + ky*transpose (rep y)
               + kz*transpose (rep z))
  let qr = sum (cos as * phiM)
  let qi = sum (sin as * phiM)
  in (qr, qi)
```

Metrics from changing a benchmark suite

Proportion of ILP problems that have less than some given number of constraints.



Number of programs: 67

Lines of code: 8621 ⇒ 8515

Change in `maps`: 467 ⇒ 213

Largest ILP size: 28104 constraints

Median ILP size: 16 constraints

Mean ILP size: 116 constraints

Mean type checking slowdown: 2.50×

Related work

- **Typed Remora**:
  - Very general/powerful; binds shape variables in types:

$$\texttt{sum} : \forall S. S\ \texttt{int} \to \texttt{int}$$

  - Inference is very difficult.

Related work

- **Typed Remora**:
  - Very general/powerful; binds shape variables in types:

    $$\texttt{sum} : \forall S. S\ \texttt{int} \to \texttt{int}$$

  - Inference is very difficult.
- **Naperian Functors** (Jeremy Gibbons):
  - Cool rank polymorphism encoding in Haskell.
  - Complicated function types (and potentially error messages).

Related work

- **Typed Remora**:
  - Very general/powerful; binds shape variables in types:

$$\texttt{sum} : \forall S. S\ \texttt{int} \to \texttt{int}$$

  - Inference is very difficult.
- **Naperian Functors** (Jeremy Gibbons):
  - Cool rank polymorphism encoding in Haskell.
  - Complicated function types (and potentially error messages).
- **Single-assignment C**:
  - Has *rank specialization* where functions have specialized definitions depending on the rank of the input.
  - No parametric polymorphism or higher-order functions.

Summary

- AUTOMAP is a conservative extension of/compatible with a Hindley-Milner-style type system for array programming.

Summary

- AUTOMAP is a conservative extension of/compatible with a Hindley-Milner-style type system for array programming.
- Anything inferred can also be inserted explicitly (much like classic type systems!)

Summary

- AUTOMAP is a conservative extension of/compatible with a Hindley-Milner-style type system for array programming.
- Anything inferred can also be inserted explicitly (much like classic type systems!)
- Type checking based on some heavy machinery (ILP), but we suspect of a fairly simple kind.

Summary

- AUTOMAP is a conservative extension of/compatible with a Hindley-Milner-style type system for array programming.
- Anything inferred can also be inserted explicitly (much like classic type systems!)
- Type checking based on some heavy machinery (ILP), but we suspect of a fairly simple kind.
- Implemented in Futhark, but not really production ready yet.
  - TODO: quality of type errors, type checking speed, better ambiguity checking.

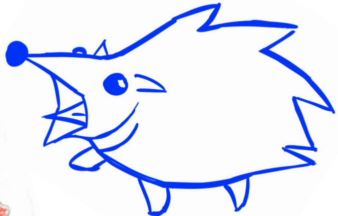
Final Things

- Check out Futhark: https://futhark-lang.org
  - There's a blog post on AUTOMAP that covers the AUTOMAP-portion of this talk in more detail.
  - The papers for each thing can also be found there, along with my PhD thesis.
- These slides and more about me at https://rschenck.com.

Final Thanks

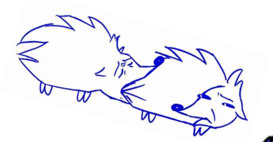
- Thanks for coming to my defense!
- Thanks to my advisors:
  - Fritz Henglein, Cosmin E. Oancea, and Troels Henriksen.
  - And everyone else at the PLTC/DIKU!
- Thanks to my commitee:
  - Michael Kirkedal for being my committee chair.
  - Sven-Bodo Scholz and Paul Kelly (and for making the trip all the way here to Copenhagen).
- And thanks to all the hedgehog artists:
  - Nikolaj Hey Hinnerskov, Fillippa Biil, Lea Henriksen, Lys Sanz Moreta, the internet, and more.


Gotta go
Fast



That's all!


FUTHARK
The problem of being faster than light is that you can only live in darkness.




GO FAST.