AUTOMAP:
Inferring Rank-Polymorphic Function Applications with Integer Linear Programming

Robert Schenck [1], Nikolaj Hey Hinnerskov [1], Troels Henriksen [1], Magnus Madsen [2], Martin Elsman [1]

[1]DIKU
University of Copenhagen
Denmark

[2]Aarhus University
Denmark

August 27th, 2024

Context

This presentation is about **Futhark**, a **functional** array language.

- Designed to study the compilation of (fast) array languages.

Context

This presentation is about **Futhark**, a **functional** array language.

- Designed to study the compilation of (fast) array languages.

- Space is function application: `f x` means $f(x)$.

Context

This presentation is about **Futhark**, a **functional** array language.

- Designed to study the compilation of (fast) array languages.
- Space is function application: `f x` means $f(x)$.
- Functions are **curried**: `f x y z` means `((f x) y) z`.

Context

This presentation is about **Futhark**, a **functional** array language.

- Designed to study the compilation of (fast) array languages.
- Space is function application: `f x` means $f(x)$.
- Functions are **curried**: `f x y z` means `((f x) y) z`.
- Hindley-Milner style **static type system** (it has parametric polymorphism).

Context

This presentation is about **Futhark**, a **functional** array language.

- Designed to study the compilation of (fast) array languages.
- Space is function application: `f x` means $f(x)$.
- Functions are **curried**: `f x y z` means `((f x) y) z`.
- Hindley-Milner style **static type system** (it has parametric polymorphism).

Example

```
def dotprod x y = sum (map (*) x y)
```

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 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 and closer to math

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

- Practically all rank polymorphic languages are **dynamic**: NumPy, APL, MATLAB, ...

`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
f: []int -> [][]int -> int
f xss xss
```

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
f: []int -> [][]int -> int
f xss xss
```

Elaborating the first application:

```
(map f xss)
```

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
f: []int -> [][]int -> int
f xss xss
```

Elaborating the first application:

```
(map f xss)
```

Elaborating the second application:

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

because

```
(map f xss) : [][][]int -> []int
```

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
f: []int -> [][]int -> int
f xss xss
```

Elaborating the first application:

```
(map f xss)
```

Elaborating the second application:

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

because

```
(map f xss) : [][][]int -> []int
```

reps can often be eliminated

```
map (\xs -> f xs xss) xss
```

Goal

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

$$\texttt{f x} \Longrightarrow \begin{cases} \texttt{map (... (map f) ...) x} \\ \text{or} \\ \texttt{f (rep ... (rep x))} \end{cases}$$

Challenge: ambiguity

Consider

```
sum: []int -> int
length : []a -> int
xss : [][]int

sum (length xss)
```

Challenge: ambiguity

Consider

```
sum: []int -> int
length : []a -> int
xss : [][]int

sum (length xss)
```

Many rank-correct elaborations:

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

Challenge: ambiguity

Consider

```
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

Consider

```
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)`

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 `maps` and `reps`.

- Generally aligns with programmer's intent/simple mental model.

The Strategy

Rule 2

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

- Generally aligns with programmer's intent/simple mental model.
- Minimization 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)`

Challenge: elaboration is global

- `sum (map length xss)` is a **global minimal** elaboration of `sum (length xss)`.
  - Inserting the **`map`** for the inner `length` application requires considering the outer `sum`.

Challenge: elaboration is global

- `sum (map length xss)` is a **global minimal** elaboration of `sum (length xss)`.
  - Inserting the **`map`** for the inner `length` application requires considering the outer `sum`.

Challenge: elaboration is global

- `sum (map length xss)` is a **global minimal** elaboration of `sum (length xss)`.
  - Inserting the **`map`** for the inner `length` application requires considering the outer `sum`.
- 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

```
f : p -> b
x : a
```

- 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.

Constraints

- Suppose

```
f : p -> b
x : a
```

- 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: $|\textbf{[ ][ ]}\texttt{int}| = 2$ and $|\texttt{int}| = 0$.

Constraints

Rank polymorphisms means rank differences are allowed.

Constraints

Rank polymorphisms means rank differences are allowed.

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

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

Constraints

Rank polymorphisms means rank differences are allowed.

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

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

  - Example:

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

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

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

M is equal to the number of **`map`**s required: **`map`** `sqrt [1,2,3]`

Constraints

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

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

Constraints

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

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

  - Example: `length : []a -> int` The application `length 3` gives the constraint

$$|\mathbf{[\,]}\,a| = R + |\texttt{int}|$$
$$1 + |a| = R \implies R = 1, 2, 3 \ldots$$

Constraints

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

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

  - Example: `length : []a -> int` The application `length 3` gives the constraint

$$|[\,]\,a| = R + |\texttt{int}|$$
$$1 + |a| = R \implies R = 1, 2, 3 \ldots$$

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

Constraints

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

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

Constraints

- Each application of a function `f : p -> 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

- Collect the constraints for each function application.

Constraints

- 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

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

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

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

Constraints

- 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

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

$$\begin{aligned}
&\text{minimize} \\
&\qquad M_1 + R_1 + M_2 + R_2 \\
&\text{subject to} \\
&\qquad \left.\begin{array}{l} M_1 + 1 + |a| = R_1 + 2 \\ M_1 = 0 \text{ or } R_1 = 0 \end{array}\right\} \texttt{length} \\
&\qquad \left.\begin{array}{l} M_2 + 1 = R_2 + M_1 \\ M_2 = 0 \text{ or } R_2 = 0 \end{array}\right\} \texttt{sum}
\end{aligned}$$

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

Constraints

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

$$
\begin{aligned}
&\text{minimize} \\
&\qquad M_1 + R_1 + M_2 + R_2 \\
&\text{subject to} \\
&\qquad \left.\begin{array}{l} M_1 + 1 + |a| = R_1 + 2 \\ M_1 = 0 \text{ or } R_1 = 0 \end{array}\right\} \texttt{length} \\
&\qquad \left.\begin{array}{l} M_2 + 1 = R_2 + M_1 \\ M_2 = 0 \text{ or } R_2 = 0 \end{array}\right\} \texttt{sum}
\end{aligned}
$$

- 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 constraint.

Automap TL;DR

1. For each application generate rank equality and Rule 1 constraint.

2. Transform constraint set into an ILP and solve.

AUTOMAP TL;DR

1. For each application generate rank equality and Rule 1 constraint.

2. Transform constraint set into an ILP and solve.

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

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

AUTOMAP TL;DR

1. For each application generate rank equality and Rule 1 constraint.

2. Transform constraint set into an ILP and solve.

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

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

4. Type check elaborated program and continue with compilation.

Formalization

Grammar

- We formalized AUTOMAP for a simple array language based on a small subset of Futhark.

Grammar

- We formalized AUTOMAP for a simple array language based on a small subset of Futhark.

$$
\begin{array}{rcll}
v & ::= & n & (n \in \mathbb{Z}) \\
 & | & \lambda x.\, e & \\
 & | & [v, \ldots, v] & \\
 & | & \mathbf{\texttt{rep}}\ v &
\end{array}
$$

Grammar

- We formalized AUTOMAP for a simple array language based on a small subset of Futhark.

$$
\begin{array}{rcll}
v & ::= & n & (n \in \mathbb{Z}) \\
 & | & \lambda x.\, e & \\
 & | & [v, \ldots, v] & \\
 & | & \mathbf{rep}\; v & \\
 & & & \\
p & ::= & \texttt{def}\; f\; x = e\; ;\; p & \\
 & | & e &
\end{array}
$$

Grammar

- We formalized AUTOMAP for a simple array language based on a small subset of Futhark.

$$
\begin{array}{rcl}
v & ::= & n \quad (n \in \mathbb{Z}) \\
 & \mid & \lambda x.\, e \\
 & \mid & [v, \ldots, v] \\
 & \mid & \mathbf{rep}\ v
\end{array}
$$

$$
\begin{array}{rcl}
p & ::= & \mathtt{def}\ f\ x = e\ ;\ p \\
 & \mid & e
\end{array}
$$

$$
\begin{array}{rcl}
e & ::= & v \\
 & \mid & x \quad (x \in V_{\mathrm{p}}) \\
 & \mid & [e, \ldots, e] \\
 & \mid & \mathbf{map}\ e\ e \\
 & \mid & \mathbf{rep}\ e \\
 & \mid & \boxed{e\ e \bigtriangleup (M, R)}
\end{array}
$$

Grammar

- We formalized AUTOMAP for a simple array language based on a small subset of Futhark.

$$
\begin{aligned}
v &::= n \quad (n \in \mathbb{Z}) \\
&\mid \lambda x.\, e \\
&\mid [v, \ldots, v] \\
&\mid \textbf{rep}\ v \\[1em]
p &::= \texttt{def}\ f\ x = e\ ;\ p \\
&\mid e
\end{aligned}
\qquad
\begin{aligned}
e &::= v \\
&\mid x \quad (x \in V_{\mathrm{p}}) \\
&\mid [e, \ldots, e] \\
&\mid \textbf{map}\ e\ e \\
&\mid \textbf{rep}\ e \\
&\mid \boxed{e\ e \bigtriangleup (M, R)}
\end{aligned}
$$

- The application $e\ e \bigtriangleup (M, R)$ is a *flexible function application*.

Flexible function applications

- During constraint generation, all function applications $f\ x$ are annotated with rank variables:

$$f\ x \quad \longrightarrow \quad f\ x \bigtriangleup (\color{blue}{M}, \color{red}{R})$$

Flexible function applications

- During constraint generation, all function applications $f\ x$ are annotated with rank variables:

$$f\ x \quad \longrightarrow \quad f\ x \bigtriangleup (M, R)$$

- $f\ x \bigtriangleup (M, R)$ will ultimately be elaborated to

$$\mathbf{map}^{s_{\mathrm{r}}(M)}\ f\ (\mathbf{rep}^{s_{\mathrm{r}}(R)}\ x)$$

  where s_{r} is the solution to the ILP and

$$\begin{aligned} \mathbf{map}^0\ e &= e \\ \mathbf{map}^{n+1}\ e &= \mathbf{map}\ (\mathbf{map}^n\ e) \end{aligned} \qquad \begin{aligned} \mathbf{rep}^0\ e &= e \\ \mathbf{rep}^{n+1}\ e &= \mathbf{rep}\ (\mathbf{rep}^n\ e) \end{aligned}$$

Languages

- The formalization is split up into three languages:

	Source lang.		
Implicit **maps**/**reps**	✓		
Explicit **maps**/**reps**	✓		
△ (M, R) annots.	✗		
Example	`sqrt` $[1, 2, 3]$		

Languages

- The formalization is split up into three languages:

	Source lang.	**Internal lang.**	
Implicit `maps`/`reps`	✓	✓	
Explicit `maps`/`reps`	✓	✓	
$\triangle$ (M, R) annots.	✗	✓	
Example	`sqrt` $[1, 2, 3]$	`sqrt` $[1, 2, 3] \triangle (M, R)$	

Languages

- The formalization is split up into three languages:

	Source lang.	**Internal lang.**	**Target lang.**
Implicit **`map`**s/**`rep`**s	✓	✓	✗
Explicit **`map`**s/**`rep`**s	✓	✓	✓
$\triangle$ (M, R) annots.	✗	✓	✗
Example	`sqrt` $[1, 2, 3]$	`sqrt` $[1, 2, 3] \triangle (M, R)$	**`map`** `sqrt` $[1, 2, 3]$

Languages

- The formalization is split up into three languages:

	Source lang.	**Internal lang.**	**Target lang.**
Implicit `maps`/`reps`	✓	✓	✗
Explicit `maps`/`reps`	✓	✓	✓
$\triangle\ (M, R)$ annots.	✗	✓	✗
Example	`sqrt` $[1, 2, 3]$	`sqrt` $[1, 2, 3] \triangle (M, R)$	`map` `sqrt` $[1, 2, 3]$

$$\textbf{Source} \xrightarrow{\text{type checking}} \textbf{Internal} \xrightarrow{\text{elaboration}} \textbf{Target}$$

Type checking

- Constraint-based type system; judgements of the form

$$\Gamma \vdash e \ :_{S} \ \sigma \ \| \ C$$

- Under environment Γ, e has scheme σ with *frame* S when the constraints in C are satisfied.

Type checking

- Constraint-based type system; judgements of the form

$$\Gamma \vdash e \;:_{S}\; \sigma \;\|\; C$$

- Under environment Γ, e has scheme σ with *frame* S when the constraints in C are satisfied.
  - Frames syntactically seperate leading dimensions that are the result of **`map`**s.
  - In general, can think of e as having the type $S\ \sigma$.

Type checking

- Constraint-based type system; judgements of the form

$$\Gamma \vdash e \ :_{S} \ \sigma \ \parallel \ C$$

- Under environment Γ, e has scheme σ with *frame* S when the constraints in C are satisfied.
  - Frames syntactically seperate leading dimensions that are the result of **map**s.
  - In general, can think of e as having the type $S\ \sigma$.
  - Example:

$$\Gamma \vdash \texttt{sqrt}\ [1,2,3] \ :_{[\,]^{M}} \ \texttt{int} \ \parallel \ \{[\,]^{M}\texttt{int} \stackrel{?}{=} [\,]\texttt{int}\}$$

Type checking

- Constraint-based type system; judgements of the form

$$\Gamma \vdash e \ :_{S}\ \sigma \ \| \ C$$

- Under environment Γ, e has scheme σ with *frame* S when the constraints in C are satisfied.
  - Frames syntactically seperate leading dimensions that are the result of **maps**.
  - In general, can think of e as having the type $S\ \sigma$.
  - Example:

$$\Gamma \vdash \texttt{sqrt}\ [1,2,3]\ :_{[\,]^M}\ \texttt{int}\ \|\ \{[\,]^M\texttt{int} \stackrel{?}{=} [\,]\texttt{int}\}$$

  - Frames are needed for proofs, improved ambiguity checking, and implementation optimizations.

Type checking rules - C-APP

- The C-APP rule types flexible function applications.

$$\frac{\Gamma \vdash e_1 \;:_{S_1}\; \tau_1 \to \tau_2 \;\|\; C_1 \qquad \Gamma \vdash e_2 \;:_{S_2}\; \tau_3 \;\|\; C_2}{}\ \text{C-APP}$$

Type checking rules - C-App

- The C-App rule types flexible function applications.

$$\dfrac{\Gamma \vdash e_1 \;:_{S_1}\; \tau_1 \to \tau_2 \;\|\; C_1 \qquad \Gamma \vdash e_2 \;:_{S_2}\; \tau_3 \;\|\; C_2 \\ M, R \text{ fresh} \qquad C = \{M \mathrel{\dot\vee} R,\ \ [\,]^{M}\ S_1\ \tau_1 \stackrel{?}{=} [\,]^{R}\ S_2\ \tau_3\}}{}\ \textsc{C-App}$$

Type checking rules - C-App

- The C-App rule types flexible function applications.

$$\dfrac{\Gamma \vdash e_1 \;:_{S_1}\; \tau_1 \to \tau_2 \;\|\; C_1 \qquad \Gamma \vdash e_2 \;:_{S_2}\; \tau_3 \;\|\; C_2 \qquad M, R \text{ fresh} \qquad C = \{M \,\dot{\vee}\, R,\; [\,]^M\, S_1\, \tau_1 \overset{?}{=} [\,]^R\, S_2\, \tau_3\}}{\Gamma \vdash e_1\, e_2 \triangle (M, R) \;:_{[\,]^M S_1}\; \tau_2 \;\|\; C \cup C_1 \cup C_2}\;\text{C-App}$$

Type checking rules - C-APP

- The C-APP rule types flexible function applications.

$$\dfrac{\Gamma \vdash e_1 \ :_{S_1}\ \tau_1 \to \tau_2 \parallel C_1 \qquad \Gamma \vdash e_2 \ :_{S_2}\ \tau_3 \parallel C_2 \qquad M, R \text{ fresh} \qquad C = \{M \dot{\vee} R,\ [\,]^M\ S_1\ \tau_1 \stackrel{?}{=} [\,]^R\ S_2\ \tau_3\}}{\Gamma \vdash e_1\ e_2 \triangle (M, R)\ :_{[\,]^M S_1}\ \tau_2 \parallel C \cup C_1 \cup C_2}\ \text{C-APP}$$

- The generated constraints are relaxed to rank constraints, which are used to form the ILP:

$$[\,]^M\ S_1\ \tau_1 \stackrel{?}{=} [\,]^R\ S_2\ \tau_3 \longrightarrow M + |S_1| + |\tau_1| \stackrel{?}{=} R + |S_2| + |\tau_3|$$

Type checking rules - C-Def

- The C-Def rule types top-level polymorphic definitions and dispatches the constraint set for each top-level definition

$$\frac{\Gamma, x : \tau \vdash e \;:\; \tau' \;\|\; C \qquad s \text{ satisfies } C \qquad \{\vec{\alpha}\} \cap \mathbf{ftv}(s(\Gamma), \sigma) = \emptyset \qquad s(\Gamma), f : \forall \vec{\alpha}. s(\tau) \to s(\tau') \vdash p \;:\; \sigma}{s(\Gamma) \vdash \texttt{def}\; f\; x = s(e)\; ;\; p \;:\; \sigma}\;\text{C-Def}$$

Type checking rules - C-DEF

- The C-DEF rule types top-level polymorphic definitions and dispatches the constraint set for each top-level definition

$$\frac{\Gamma, x : \tau \vdash e \;\mathbf{:}\; \tau' \;\|\; C \qquad s \text{ satisfies } C \qquad \{\vec{\alpha}\} \cap \mathbf{ftv}(s(\Gamma), \sigma) = \emptyset \qquad s(\Gamma), f : \forall \vec{\alpha}. s(\tau) \to s(\tau') \vdash p \;\mathbf{:}\; \sigma}{s(\Gamma) \vdash \texttt{def}\; f\; x = s(e)\;;\; p \;\mathbf{:}\; \sigma}\;\textsc{C-Def}$$

- Key idea: find a satisfying substitution for the constraint set C and apply it.

Type checking rules - C-DEF

- The C-DEF rule types top-level polymorphic definitions and dispatches the constraint set for each top-level definition

$$\frac{\Gamma, x : \tau \vdash e \ \mathbf{:}\ \tau' \ \| \ C \qquad s \text{ satisfies } C \qquad \{\vec{\alpha}\} \cap \mathrm{ftv}(s(\Gamma), \sigma) = \emptyset \qquad s(\Gamma), f : \forall \vec{\alpha}. s(\tau) \to s(\tau') \vdash p \ \mathbf{:}\ \sigma}{s(\Gamma) \vdash \texttt{def}\ f\ x = s(e)\ ;\ p \ \mathbf{:}\ \sigma} \ \text{C-DEF}$$

- Key idea: find a satisfying substitution for the constraint set C and apply it.
  - Instantiates all rank variables with integral ranks:

$$s(\texttt{sqrt}\ [1,2,3] \bigtriangleup (M, R)) \longrightarrow \texttt{sqrt}\ [1,2,3] \bigtriangleup (1, 0)$$

Type checking rules - C-DEF

- The C-DEF rule types top-level polymorphic definitions and dispatches the constraint set for each top-level definition

$$\frac{\Gamma, x : \tau \vdash e \;:\; \tau' \;\|\; C \qquad s \text{ satisfies } C \qquad \{\vec{\alpha}\} \cap \mathrm{ftv}(s(\Gamma), \sigma) = \emptyset \qquad s(\Gamma), f : \forall\vec{\alpha}.s(\tau) \to s(\tau') \vdash p \;:\; \sigma}{s(\Gamma) \vdash \texttt{def}\; f\; x = s(e)\;;\; p \;:\; \sigma}\;\textsc{C-Def}$$

- Key idea: find a satisfying substitution for the constraint set C and apply it.
  - Instantiates all rank variables with integral ranks:

$$s(\texttt{sqrt}\;[1,2,3] \bigtriangleup (M, R)) \longrightarrow \texttt{sqrt}\;[1,2,3] \bigtriangleup (1, 0)$$

  - s also substitutes type variables.

SOLVE: Constraint set solving algorithm

input : A constraint set C.
output: A satisfying substitution s.

$|C| \leftarrow$ construct the associated rank constraint set from C;
$I \leftarrow$ construct the corresponding ILP from $|C|$;
$s_{\mathrm{r}} \leftarrow$ solve I using an ILP solver;
if s_{r} **then**
    $I' \leftarrow$ add constraints to I to ban solution s_{r} and enforce same size;
    $s'_{\mathrm{r}} \leftarrow$ solve I';
    if s'_{r} **then**
        return $\perp$;   `//` $|C|$ `is ambiguous; fail`
    else
        return $\textsc{Unify}(s_{\mathrm{r}}(C)) \circ s_{\mathrm{r}}$
else
    return $\perp$

SOLVE: Constraint set solving algorithm

input : A constraint set C.
output: A satisfying substitution s.

$|C| \leftarrow$ construct the associated rank constraint set from C;

$$[\,]^{M}\ S_1\ \tau_1 \stackrel{?}{=} [\,]^{R}\ S_2\ \tau_3 \longrightarrow M + |S_1| + |\tau_1| \stackrel{?}{=} R + |S_2| + |\tau_3|;$$

$s_{\mathrm{r}} \leftarrow$ solve I using an ILP solver;
if s_{r} **then**
    $I' \leftarrow$ add constraints to I to ban solution s_{r} and enforce same size;
    $s'_{\mathrm{r}} \leftarrow$ solve I';
    if s'_{r} **then**
        return $\bot$;   `// |C| is ambiguous; fail`
    else
        return UNIFY$(s_{\mathrm{r}}(C)) \circ s_{\mathrm{r}}$
else
    return $\bot$

SOLVE: Constraint set solving algorithm

input : A constraint set C.
output: A satisfying substitution s.

$|C| \leftarrow$ construct the associated rank constraint set from C;
$I \leftarrow$ construct the corresponding ILP from $|C|$;
$s_{\mathrm{r}} \leftarrow$ solve I using an ILP solver;
if s_{r} **then**
    $I' \leftarrow$ add constraints to I to ban solution s_{r} and enforce same size;
    $s'_{\mathrm{r}} \leftarrow$ solve I';
    if s'_{r} **then**
        return $\perp$; `// |C| is ambiguous; fail`
    else
        return $\textsc{Unify}(s_{\mathrm{r}}(C)) \circ s_{\mathrm{r}}$
else
    return $\perp$

SOLVE: Constraint set solving algorithm

input : A constraint set C.
output: A satisfying substitution s.

$|C| \leftarrow$ construct the associated rank constraint set from C;
$I \leftarrow$ construct the corresponding ILP from $|C|$;
$s_{\mathrm{r}} \leftarrow$ solve I using an ILP solver;
if s_{r} **then**
  $I' \leftarrow$ add constraints to I to ban solution s_{r} and enforce same size;
  $s'_{\mathrm{r}} \leftarrow$ solve I';
  if s'_{r} **then**
    return $\bot$; // $|C|$ is ambiguous; fail
  else
    return $\textsc{Unify}(s_{\mathrm{r}}(C)) \circ s_{\mathrm{r}}$
else
  return $\bot$

SOLVE: Constraint set solving algorithm

input : A constraint set C.
output: A satisfying substitution s.

$|C| \leftarrow$ construct the associated rank constraint set from C;
$I \leftarrow$ construct the corresponding ILP from $|C|$;
$s_{\mathrm{r}} \leftarrow$ solve I using an ILP solver;
if s_{r} **then**
    $I' \leftarrow$ add constraints to I to ban solution s_{r} and enforce same size;
    $s'_{\mathrm{r}} \leftarrow$ solve I';
    if s'_{r} **then**
        return $\perp$;   // $|C|$ is ambiguous; fail
    else
        return UNIFY$(s_{\mathrm{r}}(C)) \circ s_{\mathrm{r}}$
else
    return $\perp$

Solve: Constraint set solving algorithm

input : A constraint set C.
output: A satisfying substitution s.

$|C| \leftarrow$ construct the associated rank constraint set from C;
$I \leftarrow$ construct the corresponding ILP from $|C|$;
$s_{\mathrm{r}} \leftarrow$ solve I using an ILP solver;
if s_{r} **then**
  $I' \leftarrow$ add constraints to I to ban solution s_{r} and enforce same size;
  $s'_{\mathrm{r}} \leftarrow$ solve I';
  if s'_{r} **then**
    return $\perp$; // $|C|$ is ambiguous; fail
  else
    return $\textsc{Unify}(s_{\mathrm{r}}(C)) \circ s_{\mathrm{r}}$
else
  return $\perp$

SOLVE: Constraint set solving algorithm

input : A constraint set C.
output: A satisfying substitution s.

$|C| \leftarrow$ construct the associated rank constraint set from C;
$I \leftarrow$ construct the corresponding ILP from $|C|$;
$s_{\mathrm{r}} \leftarrow$ solve I using an ILP solver;
if s_{r} **then**
- $I' \leftarrow$ add constraints to I to ban solution s_{r} and enforce same size;
- $s'_{\mathrm{r}} \leftarrow$ solve I';
- **if** s'_{r} **then**
  - **return** $\perp$; // $|C|$ is ambiguous; fail
- **else**
  - **return** UNIFY$(s_{\mathrm{r}}(C)) \circ s_{\mathrm{r}}$

else
- **return** $\perp$

Solve: Constraint set solving algorithm

input : A constraint set C.
output: A satisfying substitution s.

$|C| \leftarrow$ construct the associated rank constraint set from C;
$I \leftarrow$ construct the corresponding ILP from $|C|$;
$s_{\mathrm{r}} \leftarrow$ solve I using an ILP solver;
if s_{r} **then**
  $I' \leftarrow$ add constraints to I to ban solution s_{r} and enforce same size;
  $s'_{\mathrm{r}} \leftarrow$ solve I';
  if s'_{r} **then**
    return $\perp$; // $|C|$ is ambiguous; fail
  else
    return $\textsc{Unify}(s_{\mathrm{r}}(C)) \circ s_{\mathrm{r}}$
else
  return $\perp$

SOLVE: Constraint set solving algorithm

input : A constraint set C.
output: A satisfying substitution s.

$|C| \leftarrow$ construct the associated rank constraint set from C;

$I \leftarrow$ construct the corresponding ILP from $|C|$;

$s_{\mathrm{r}} \leftarrow$ solve I using an ILP solver;

if s_{r} **then**

— $I' \leftarrow$ add constraints to I to ban solution s_{r} and enforce same size;

— $s'_{\mathrm{r}} \leftarrow$ solve I';

— **if** s'_{r} **then**

— — **return** $\perp$; // $|C|$ is ambiguous; fail

— **else**

— — **return** UNIFY($s_{\mathrm{r}}(C)$) $\circ\, s_{\mathrm{r}}$

else

— **return** $\perp$

SOLVE: Constraint set solving algorithm

input : A constraint set C.
output: A satisfying substitution s.

$|C| \leftarrow$ construct the associated rank constraint set from C;
$I \leftarrow$ construct the corresponding ILP from $|C|$;
$s_{\mathrm{r}} \leftarrow$ solve I using an ILP solver;
if s_{r} **then**
    $I' \leftarrow$ add constraints to I to ban solution s_{r} and enforce same size;
    $s'_{\mathrm{r}} \leftarrow$ solve I';
    if s'_{r} **then**
        return $\perp$; // $|C|$ is ambiguous; fail
    else
        return $\textsc{Unify}(s_{\mathrm{r}}(C)) \circ s_{\mathrm{r}}$
else
    return $\perp$

SOLVE: Constraint set solving algorithm

input : A constraint set C.
output: A satisfying substitution s.

$|C| \leftarrow$ construct the associated rank constraint set from C;
$I \leftarrow$ construct the corresponding ILP from $|C|$;
$s_{\mathrm{r}} \leftarrow$ solve I using an ILP solver;
if s_{r} **then**
    $I' \leftarrow$ add constraints to I to ban solution s_{r} and enforce same size;
    $s'_{\mathrm{r}} \leftarrow$ solve I';
    if s'_{r} **then**
        return $\perp$; // $|C|$ is ambiguous; fail
    else
        return $\textsc{Unify}(s_{\mathrm{r}}(C)) \circ s_{\mathrm{r}}$
else
    return $\perp$

Properties of SOLVE

- All satisfiers s can be decomposed into a type and rank substitution:

Proposition

If s satisfies C, there exists a rank substitution s_{r} that satisfies $|C|$ and there exists a closed type substitution s_{t} such that $s|_{\mathrm{ftv}(C)\cup\mathrm{frv}(C)} = s_{\mathrm{t}} \circ s_{\mathrm{r}}$.

Properties of SOLVE

- All satisfiers s can be decomposed into a type and rank substitution:

Proposition

If s satisfies C, there exists a rank substitution s_{r} that satisfies $|C|$ and there exists a closed type substitution s_{t} such that $s|_{\mathrm{ftv}(C)\cup\mathrm{frv}(C)} = s_{\mathrm{t}} \circ s_{\mathrm{r}}$.

- You can always build a satisfier $s_{\mathrm{t}} \circ s_{\mathrm{r}}$ of C from a satisfier s_{r} of $|C|$:

Proposition

If C is satisfiable and s_{r} satisfies $|C|$ then there is a closed type substitution s_{t} such that the substitution $s = s_{\mathrm{t}} \circ s_{\mathrm{r}}$ satisfies C.

Elaboration

- Reminder: a satisfying substitution s for a constraint set C (which SOLVE gives us) instantiates all rank variables with integral ranks:

$$s(\texttt{sqrt}\ [1,2,3] \bigtriangleup (M, R)) \longrightarrow \texttt{sqrt}\ [1,2,3] \bigtriangleup (1, 0)$$

Elaboration

- Reminder: a satisfying substitution s for a constraint set C (which SOLVE gives us) instantiates all rank variables with integral ranks:

$$s(\texttt{sqrt}\ [1, 2, 3] \mathbin{\triangle} (M, R)) \longrightarrow \texttt{sqrt}\ [1, 2, 3] \mathbin{\triangle} (1, 0)$$

- How do we then transform such an expression into one with explicit **`map`**s and **`rep`**s?

$$\texttt{sqrt}\ [1, 2, 3] \mathbin{\triangle} (1, 0) \longrightarrow \textbf{map}\ \texttt{sqrt}\ [1, 2, 3]$$

Elaboration

- Reminder: a satisfying substitution s for a constraint set C (which SOLVE gives us) instantiates all rank variables with integral ranks:

$$s(\texttt{sqrt}\ [1,2,3] \bigtriangleup (M,R)) \longrightarrow \texttt{sqrt}\ [1,2,3] \bigtriangleup (1,0)$$

- How do we then transform such an expression into one with explicit **`map`**s and **`rep`**s?

$$\texttt{sqrt}\ [1,2,3] \bigtriangleup (1,0) \longrightarrow \textbf{map}\ \texttt{sqrt}\ [1,2,3]$$

- The AM transformation converts from the internal language to the target language:

$$\begin{array}{lcl}
\mathrm{AM}([e_1,\ldots,e_m]) & = & [\mathrm{AM}(e_1),\ldots,\mathrm{AM}(e_m)] \\
\mathrm{AM}(\texttt{def}\ f\ x = e\ ;\ p) & = & \texttt{def}\ f\ x = \mathrm{AM}(e)\ ;\ \mathrm{AM}(p) \\
\mathrm{AM}(e_1\ e_2 \bigtriangleup (n_{\mathrm{M}}, n_{\mathrm{R}})) & = & \boxed{\textbf{map}^{n_{\mathrm{M}}}\ \mathrm{AM}(e_1)\ (\textbf{rep}^{n_{\mathrm{R}}}\ \mathrm{AM}(e_2))} \\
 & \vdots &
\end{array}$$

AM Properties

- The AM transformation preserves well-typedness of programs.

Proposition: Well-typedness

If $\Gamma \vdash e \;:_S\; \sigma \;\|\; C$ and s is a satisfier of C, then $s(\Gamma) \vdash \mathrm{AM}(s(e)) \;:\; s(S\,\sigma)$.

AM Properties

- The AM transformation preserves well-typedness of programs.

Proposition: Well-typedness

If $\Gamma \vdash e \;:_S\; \sigma \;\|\; C$ and s is a satisfier of C, then $s(\Gamma) \vdash \mathrm{AM}(s(e)) \;:\; s(S\,\sigma)$.

Proposition: Well-typedness

If $\Gamma \vdash p \;:\; \sigma$ then $\Gamma \vdash \mathrm{AM}(p) \;:\; \sigma$.

Consistency properties

Forward Consistency

If the programmer inserts an otherwise inferred `map` or `rep` operation then the resulting program is unambiguous and its elaboration is semantically equivalent to the elaboration of the original program.

Consistency properties

Forward Consistency

If the programmer inserts an otherwise inferred **map** or **rep** operation then the resulting program is unambiguous and its elaboration is semantically equivalent to the elaboration of the original program.

Hindley-Milner	**AUTOMAP**	
$\Gamma \vdash e : \sigma \;:\; \sigma$	$\Gamma \vdash \mathtt{map}\, f\, x \;:\; \sigma$	$\Gamma \vdash f\, (\mathtt{rep}\, x) \;:\; \sigma$
$\uparrow$	$\uparrow$	$\uparrow$
$\Gamma \vdash e \;:\; \sigma$	$\Gamma \vdash f\, x \;:\; \sigma$	$\Gamma \vdash f\, x \;:\; \sigma$

Backwards Consistency

If the programmer removes an explicit `map` or `rep` operation then the resulting program is *either* ambiguous or unambiguous and its elaboration is semantically equivalent to the elaboration of the original program.

Backwards Consistency

If the programmer removes an explicit **map** or **rep** operation then the resulting program is *either* ambiguous or unambiguous and its elaboration is semantically equivalent to the elaboration of the original program.

Hindley-Milner	**AUTOMAP**	
$\Gamma \vdash e\!: \sigma \;:\; \sigma$	$\Gamma \vdash \mathbf{map}\; f\; x \;:\; \sigma$	$\Gamma \vdash f\; (\mathbf{rep}\; x) \;:\; \sigma$
$\downarrow$	$\downarrow$	$\downarrow$
$\Gamma \vdash e \;:\; \sigma$	$\Gamma \vdash f\; x \;:\; \sigma$	$\Gamma \vdash f\; x \;:\; \sigma$

Backwards Consistency

If the programmer removes an explicit `map` or `rep` operation then the resulting program is *either* ambiguous or unambiguous and its elaboration is semantically equivalent to the elaboration of the original program.

Hindley-Milner	**AUTOMAP**	
$\Gamma \vdash e : \sigma \;\mathbf{:}\; \sigma$	$\Gamma \vdash \texttt{map}\; f\; x \;\mathbf{:}\; \sigma$	$\Gamma \vdash f\; (\texttt{rep}\; x) \;\mathbf{:}\; \sigma$
$\Updownarrow$	$\Updownarrow$	$\Updownarrow$
$\Gamma \vdash e \;\mathbf{:}\; \sigma$	$\Gamma \vdash f\; x \;\mathbf{:}\; \sigma$	$\Gamma \vdash f\; x \;\mathbf{:}\; \sigma$

Implementation

- Implemented in the Futhark compiler.

Implementation

- Implemented in the Futhark compiler.
- Four phases:

Constraint gen. → ILP solving → Residual solving → Elaboration

Implementation

- Implemented in the Futhark compiler.
- Four phases:

  Constraint gen. → ILP solving → Residual solving → Elaboration

- Aside from size inference, works very similarly to the SOLVE algorithm, with a couple of exceptions.

Implementation - induced **reps**

- $\texttt{map}\ (\lambda y.\, xs \ast y)\ ys$ can be elaborated to

$$\texttt{map}\,(\lambda y.\,\textbf{map}\,(\ast)\,xs\,(\textbf{rep}\,y))\,ys \quad (1) \quad \text{or} \quad \texttt{map}\,(\lambda y.\,\textbf{map}\,(\ast)\,xs\,y)\,(\textbf{rep}\,ys) \quad (2)$$

Implementation - induced **reps**

- $\mathtt{map}\ (\lambda y.\, xs\ *\ y)\ ys$ can be elaborated to

  $\mathtt{map}\,(\lambda y.\,\mathbf{map}\,(*)\,xs\,(\mathbf{rep}\,y))\,ys \quad (1) \quad \text{or} \quad \mathtt{map}\,(\lambda y.\,\mathbf{map}\,(*)\,xs\,y)\,(\mathbf{rep}\,ys) \quad (2)$

- Both have size **2** and are minimal.

Implementation - induced `rep`s

- $\texttt{map}\ (\lambda y.\, xs * y)\ ys$ can be elaborated to

$$\texttt{map}\,(\lambda y.\,\textbf{map}\,(*)\,xs\,(\textbf{rep}\,y))\,ys \quad (1) \quad \text{or} \quad \texttt{map}\,(\lambda y.\,\textbf{map}\,(*)\,xs\,y)\,(\textbf{rep}\,ys) \quad (2)$$

- Both have size **2** and are minimal.
- The `rep` in (1) is **induced** by the outer **map**. It will be eliminated by `rep` fusion:

$$\texttt{map}\,(\lambda y.\,\textbf{map}\,(*)\,xs\,(\textbf{rep}\,y))\,ys \quad \longrightarrow \quad \texttt{map}\,(\lambda y.\,\textbf{map}\,(\lambda x.\,x * y)\,xs)\,ys$$

Implementation - induced $\texttt{\textbf{rep}}$s

- $\texttt{map}\ (\lambda y.\, xs \ast y)\ ys$ can be elaborated to

$$\texttt{map}\,(\lambda y.\,\texttt{\textbf{map}}\,(\ast)\,xs\,(\texttt{\textbf{rep}}\,y))\,ys \quad (1) \quad \text{or} \quad \texttt{map}\,(\lambda y.\,\texttt{\textbf{map}}\,(\ast)\,xs\,y)\,(\texttt{\textbf{rep}}\,ys) \quad (2)$$

- Both have size **2** and are minimal.
- The $\texttt{\textbf{rep}}$ in (1) is **induced** by the outer **map**. It will be eliminated by $\texttt{\textbf{rep}}$ fusion:

$$\texttt{map}\,(\lambda y.\,\texttt{\textbf{map}}\,(\ast)\,xs\,(\texttt{\textbf{rep}}\,y))\,ys \quad \longrightarrow \quad \texttt{map}\,(\lambda y.\,\texttt{\textbf{map}}\,(\lambda x.\,x \ast y)\,xs)\,ys$$

- $\texttt{map}\,(\lambda y.\,\texttt{\textbf{map}}\,(\lambda x.\,x \ast y)\,xs)\,ys$ has size **1**.
- $\Rightarrow$ we can disambiguate by only counting **non-induced** $\texttt{\textbf{rep}}$S.

Implementation - induced `rep`s

- Frames track the number of `map`s in an application ($S = [\,]^M S_1$):

$$\Gamma \vdash e_1\, e_2 \triangle (M, R) \;:_S\; \sigma \;\|\; C$$

Implementation - induced `rep`s

- Frames track the number of `map`s in an application ($S = [\,]^{M} S_1$):

$$\Gamma \vdash e_1\, e_2 \triangle (M, R) \;:_S\; \sigma \;\|\; C$$

- The number of non-induced `rep`s is found by subtracting the rank of the frame:

$$\#\text{ non-induced } \texttt{rep}\text{s} = \max(0, |R| - |S|)$$

Implementation - induced `reps`

- Frames track the number of `map`s in an application ($S = [\,]^{M} S_1$):

$$\Gamma \vdash e_1\, e_2 \bigtriangleup (M, R) \;:_S\; \sigma \;\parallel\; C$$

- The number of non-induced `rep`s is found by subtracting the rank of the frame:

$$\#\ \text{non-induced } \texttt{reps} = \max(0, |R| - |S|)$$

- ILP objective becomes

$$\dots + M + \max(0, |R| - |S|) + \dots$$

User experience

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

User experience

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

- Ambiguity feedback:
  `sum (length xss)` can be elaborated to:
  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 Futhark functions.
  - Programmer free to use AUTOMAP to whatever extent they wish.

- Ambiguity feedback:
  `sum (length xss)` can be elaborated to:
  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 Futhark functions.
  - Programmer free to use AUTOMAP to whatever extent they wish.

- Ambiguity feedback:
  `sum (length xss)` can be elaborated to:
  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: compiler can always elaborate any implicit `map`s or `rep`s.

User experience

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

- Ambiguity feedback:
  `sum (length xss)` can be elaborated to:
  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: compiler can always elaborate any implicit `map`s or `rep`s.

Practical impact

- 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

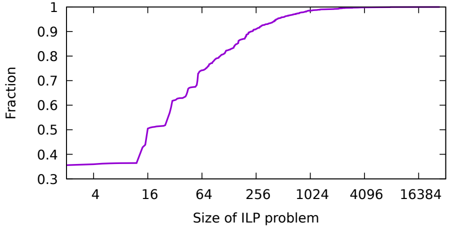
```
def main [nK][nX]
         (kx: [nK]f32) (ky: [nK]f32) (kz: [nK]f32)
         (x: [nX]f32) (y: [nX]f32) (z: [nX]f32)
         (phiR: [nK]f32) (phiI: [nK]f32)
       : ([nX]f32, [nX]f32) =
  let phiM = map (\r i -> r*r + i*i) phiR phiI
  let as = map (\x_e y_e z_e ->
          map (2*pi*)
            (map (\kx_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 (\a -> sum(map2 (*) phiM (map cos a))) as
  let qi = map (\a -> 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:

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

  - 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 type system for array programming.

Summary

- AUTOMAP is a conservative extension of/compatible with a Hindley-Milner 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 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 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, alternate ambiguity checking.

- Check out Futhark: https://futhark-lang.org
  - There's a blog post on AUTOMAP that covers much of this talk in more detail.
  - An AUTOMAP paper will be published at OOPSLA 2024, preprint available at https://rschenck.com.