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

October 23th, 2024

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]`

- 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
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)
```

reps can often be eliminated

```
map (\xs -> 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**.

$$\texttt{f x} \Longrightarrow \texttt{map (... (map f) ...) (rep ... (rep x) ...)}$$

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 `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.

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))`

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)`

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

```
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, $|[\,][\,]\texttt{int}| = 2$ 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

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

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

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

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

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

Constraints - `rep`

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

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

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

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

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

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

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

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

  - 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 -> 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)`

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

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

Constraints to ILPs

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

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

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

- 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)`

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

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

Constraints 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
- 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 $R_i = 0$:

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

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 $R_i = 0$:

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

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

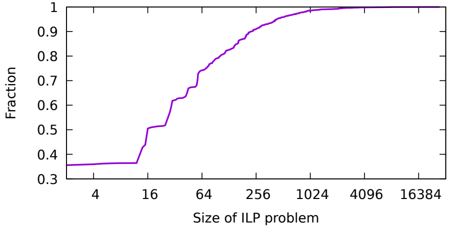
```
def main [nK][nX]
        (kx: [nK]f32) (ky: [nK]f32) (kz: [nK]f32)
        (x: [nX]f32) (y: [nX]f32) (z: [nX]f32)
        (phiR: [nK]f32) (phiI: [nK]f32)
      : ([nX]f32, [nX]f32) =
  let phiM = map (\r i -> r*r + i*i) phiR phiI
  let as = map (\x_e y_e z_e ->
          map (2*pi*)
           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

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.

That's it!

- Check out Futhark: https://futhark-lang.org
  - There's a blog post on AUTOMAP that covers this talk in more detail.
  - The paper with a full formalization can also be found there.
- These slides and more about me at https://rschenck.com.