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

June 25th, 2024

Context

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

- Designed to study compilation of array languages.
- Space is function application: `f x` means $f(x)$.

Context

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

- Designed to study compilation of 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 compilation of 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
```

rep can often be free by fusing it into function

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

Goal

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

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

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`
  - `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 `maps` and `reps`.

The Strategy

Rule 2

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

- 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` 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` requires considering the outer `sum`.

- To find minimal elaborations, must consider all applications **simultaneously**.

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
```

- 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: $|$**[ ][ ]**`int`$| = 2$ and $|$`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)`

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

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

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

Constraints

- 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

- 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 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.

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

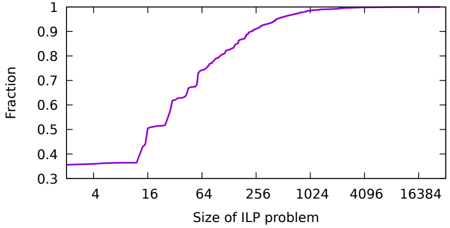
```
def main [nK][nX]
         (kx: [nK]f32) (ky: [nK]f32) (kz: [nK]f32)
         (x: [nX]f32) (y: [nX]f32) (z: [nX]f32)
         (phiR: [nK]f32) (phiI: [nK]f32)
       : ([nX]f32, [nX]f32) =
  let phiM = map (\r i -> r*r + i*i) phiR phiI
  let as = map (\x_e y_e z_e ->
                 map (2*pi*)
                   (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: $\forall S.S\ \texttt{int} \to \texttt{int}$.
  - Inference is very difficult.

Related work

- **Typed Remora**:
  - Very general/powerful; binds shape variables in types: $\forall S.S\ \texttt{int} \to \texttt{int}$.
  - 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: $\forall S.S$ `int` $\rightarrow$ `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.

- Check out Futhark: https://futhark-lang.org
  - There's a blog post on AUTOMAP that covers this talk in more detail. (A paper is coming too.)

- Check out me: https://rschenck.com
  - Graduating December 2024, **looking for a postdoc** (or a real job) in types/functional programming/compilers.

Troubles

Can we always rewrite `map f x` as `f x`?

- Consider (outer product)

```
map (\x -> map (*x) ys) xs
```

Troubles

Can we always rewrite `map f x` as `f x`?

- Consider (outer product)

```
map (\x -> map (*x) ys) xs
```

- Removing the innermost `map` works fine:

```
map (\x -> x * ys) xs
```

Troubles

Can we always rewrite `map f x` as `f x`?

- Consider (outer product)

```
map (\x -> map (*x) ys) xs
```

- Removing the innermost `map` works fine:

```
map (\x -> x * ys) xs
```

- Removing the outer `map`:

```
(\x -> x * ys) xs
```

(which is the same as `xs * ys`). This is elaborated by AUTOMAP to

```
map (*) xs ys
```