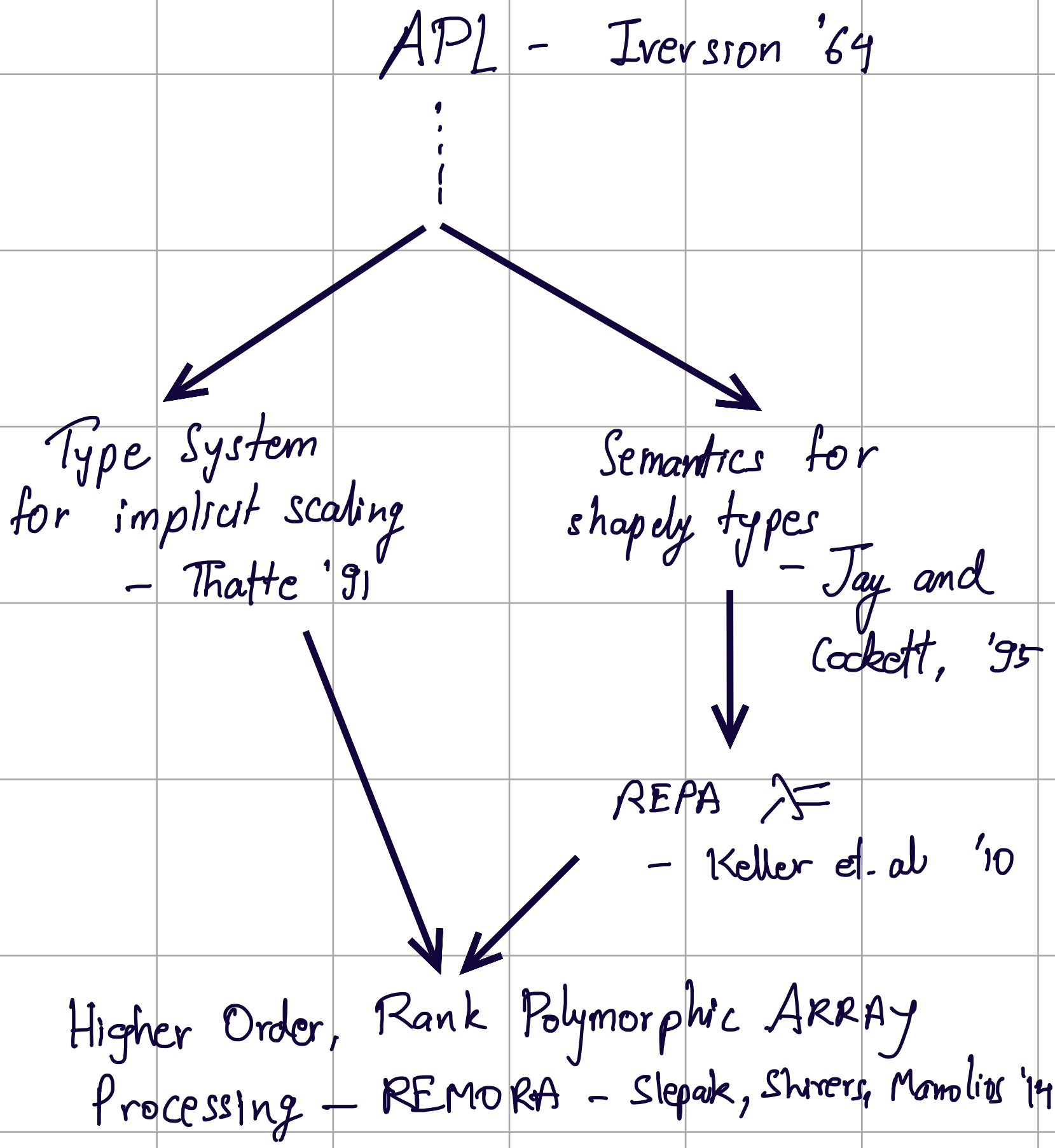
History of Rank Polymorphic Type Systems in Array Programming Languages



A Programming Language

- Kenneth E. Iverson was a mathematician who was interested in notations for mathematical thinking.

  [Notation as a tool of thought]

- In late 1950s, developed notations for data processing.

- Went on to work @ IBM, where he extended his notations to also describe systems and algorithms.

"Iverson's Better Math"

IBM SELECTRIC TYPEWRITER

[source: Wikipedia]

A selectric typing element →

IBM typewheel and typeball containing greek APL characters

* Since the notation was developed as a means of teaching and interpersonnel communication, it consisted of normal MATH symbols.

For example:

- 15

  1 2 3 4 5

- +/R

  15

- R ← 15

  R ← 15

- 2↓R

  3 4 5

Each primitive can be used in
- monadic form
- dyadic form

* 1

2.718281828

* 2.718281826

0.999999998

- 1.2 + 2.3

  3.5

- 3 × 4 + 5

  27

- (3 × 4) + 5

  17

• 1101∧1011

1001

• 1101∧0

0000

• ⍴⌊5 • 33⍴⌊5

5

1 2 3
4 5 1
2 3 4

• (33⍴1)× 33⍴⌊9

?

• ⌈⌊3

3

Let $M = \begin{matrix} 1 & 2 \\ 3 & 4 \end{matrix}$, $N = \begin{matrix} 5 & 6 \\ 7 & 8 \end{matrix}$

- `M +.× N`

$$\begin{matrix} 19 & 22 \\ 43 & 50 \end{matrix}$$

- `M ⌈.+ N` ?

- `(⍳3) ∘.× (⍳3)`

$$\begin{matrix} 1 & 2 & 3 \\ 2 & 4 & 6 \\ 3 & 6 & 9 \end{matrix}$$

- `(⍳3) ∘.= (⍳3)` ?

$$(\sim R \in R \; 0.\times \; R) \; / R \leftarrow 1 \downarrow \iota 100$$

$$R = [2, 3 \ldots 100]$$

$$\text{not} \left([2,3..100] \in \begin{bmatrix} 4 & 6 & 8 \ldots & 200 \\ 6 & 9 & \ldots & 300 \\ 200 & 300 & \ldots & 10000 \end{bmatrix} \right)$$

$$\downarrow$$

$$[1, 1, 0, 1, 0, 1, 0, 0, 0, 1 \ldots] \; / \; R$$

Concise, clear description

but pays a high cost of complexity

...an intellectual bottleneck that has kept us tied to word-at-a-time thinking instead of encouraging us to think in terms of the larger conceptual units of the task at hand. Thus programming is basically planning and detailing the enormous traffic of words through the von Neumann bottleneck...

-J. Backus in his 1977 Turing speech

APL PROS:

1. Language built for MATHS.

2. Superb treatment of homogenous arrays of reals.

3. Shows a solution to the "Von-Neumann Bottleneck." Uses aggregate operations instead of iteration or recursion. Compare

B ← A × n instead of

```
for (i = 0; i < sizeof(A)/sizeof(A[0]); i++)
    for (j=0; j < sizeof(A[0])/sizeof(A[0][0]);
                                        j++)
        .
        .
        .
            B[i][j]... = A[i][j] * n
```

scalar operation

- ARRAY PROGRAMMING was huge in the 70s.
- APL was considered extremely important.
- Iverson was awarded the Turing Award in 1979 for designing APL, and also received a mention in Backus' 1977 Turing Speech

Limitations of APL :

- Limited function arities:
  niladic, monadic or dyadic

- Ad-hoc semantics:

```
      +/ ⍳4
10
      +/ 0⍴0
1
```

```
      Sum ← {⍺+⍵}
      Sum/ ⍳4
10
      Sum/ 0⍴0
DOMAIN ERROR
```

- No support for Data Structures other than Arrays, or even heterogenous or sparse arrays
- Complex semantics:

$$\begin{bmatrix}[1 & 2] \\ [3 & 4]\end{bmatrix} + \begin{bmatrix}[5 & 6] \\ [7 & 8]\end{bmatrix}$$

$$\begin{bmatrix}[6 & 8] \\ [10 & 12]\end{bmatrix}$$

$$\begin{bmatrix}[1 & 2] \\ [3 & 4]\end{bmatrix} ++ \begin{bmatrix}[5 & 6] \\ [7 & 8]\end{bmatrix}$$

(A)
$$\begin{bmatrix}[1 & 2] \\ [3 & 4] \\ [5 & 6] \\ [7 & 8]\end{bmatrix}$$

(B)
$$\begin{bmatrix}[1 & 2 & 5 & 6] \\ [3 & 4 & 7 & 8]\end{bmatrix}$$

- Complex semantics:

$$\begin{bmatrix}[1 & 2] \\ [3 & 4]\end{bmatrix} + \begin{bmatrix}[5 & 6] \\ [7 & 8]\end{bmatrix}$$

$$\begin{bmatrix}[6 & 8] \\ [10 & 12]\end{bmatrix}$$

$$\begin{bmatrix}[1 & 2] \\ [3 & 4]\end{bmatrix} ++ \begin{bmatrix}[5 & 6] \\ [7 & 8]\end{bmatrix}$$

(B)
$$\begin{bmatrix}[1 & 2 & 5 & 6] \\ [3 & 4 & 7 & 8]\end{bmatrix}$$

- Expressiveness:

APL: $a + b \times V$ } great for mathematicians

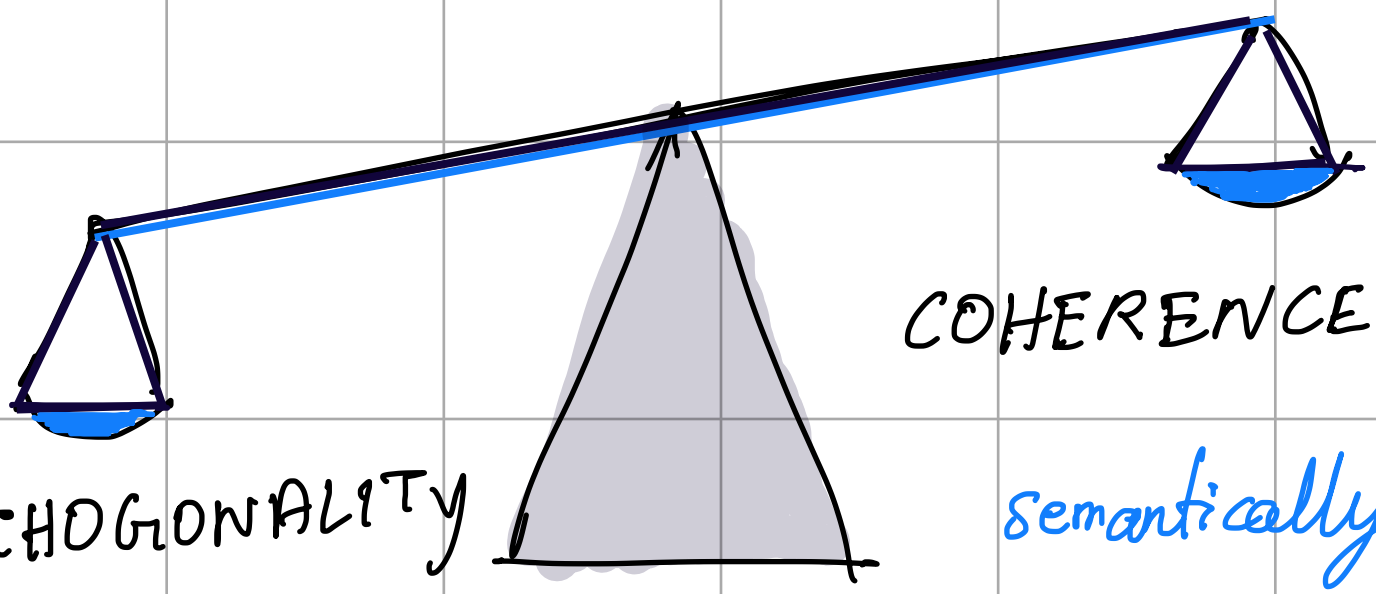
ML:

```
map (op+) (distl (a, map (op x) (distl (b, V))))
```

great for an optimising compiler

Tons of people walked away as there was no compiler support for APL.

TYPE SYSTEM FOR IMPLICIT SCALING – THATTE '90



treat data and functions as first class citizens

semantically unique coercion for each subtyping relationship.

e

$e ::= x \mid \lambda x_\tau . e \mid e_1 e_2$

$\mid e_1, e_2 \mid e \downarrow i \ (i = 1 \text{ or } 2) \mid nil_\tau$

$\mid e_1 :: e_2 \mid hd\ e \mid tl\ e$

abbreviate $[e_1, e_2 \ldots e_n]$ for

$e_1 :: (e_2 :: (e_3 \ldots :: nil))$

τ

$\tau ::= \iota \mid \tau_1 \times \tau_2 \mid [\tau] \mid \tau_1 \to \tau_2$

$e \longrightarrow\!\!\!\!\!\rightarrow e'$

e is expected to be coerced to e' by a minimal typing derivation.

eg:

- Square $[1,2,3] \longrightarrow\!\!\!\!\!\rightarrow (\propto \text{square})\ [1,2,3]$

  $= [1,4,9]$

- $1 + [1,2,3] \longrightarrow\!\!\!\!\!\rightarrow (\propto +)\ (\text{distl}\ (1, [1,2,3]))$

  $= (\propto +)\ [(1,1), (1,2), (1,3)]$

  $= [2,3,4]$

- $[1,2,3] + [2,3,4] \longrightarrow\!\!\!\!\!\rightarrow (\propto +)\ (\text{trans}\ ([1,2,3], [2,3,4]))$

  $= (\propto +)\ ([(1,2), (2,3), (3,4)])$

  $= [3,5,7]$

$$\vdash \tau_1 \lesssim \tau_2 \Rightarrow f$$

SCL : $\vdash \tau_1 \to \tau_2 \lesssim [\tau_1] \to [\tau_2] \Rightarrow \lambda$

ZIP : $\vdash [\tau_1] \times [\tau_2] \to [\tau_1 \times \tau_2] \Rightarrow trans$

REPL : $\vdash \tau_1 \times [\tau_2] \to [\tau_1 \times \tau_2] \Rightarrow distl$

REPR : $\vdash [\tau_1] \times \tau_2 \to [\tau_1 \times \tau_2] \Rightarrow distr$

coherence problem

$[int] \times [int] \lesssim [[int] \times int] \lesssim [[int \times int]]$

distr . distl

$[int] \times [int] \lesssim [int \times [int]] \lesssim [[int \times int]]$

distl . distr

$$\vdash \tau_1 \lesssim \tau_2 \Rightarrow f$$

SCL : $\vdash \tau_1 \to \tau_2 \lesssim [\tau_1] \to [\tau_2] \Rightarrow \lambda$

ZIP : $\vdash [\tau_1] \times [\tau_2] \to [\tau_1 \times \tau_2] \Rightarrow trans$

REPL : $\vdash \iota_1 \times [\tau_2] \to [\iota_1 \times \tau_2] \Rightarrow distL$

REPR : $\vdash [\tau_1] \times \iota_2 \to [\tau_1 \times \iota_2] \Rightarrow distr$

$\vdash \tau_1 \lesssim \tau_2 \Rightarrow f$

SCL : $\vdash \tau_1 \to \tau_2 \lesssim [\tau_1] \to [\tau_2] \Rightarrow \alpha$

ZIP : $\vdash [\tau_1] \times [\tau_2] \to [\tau_1 \times \tau_2] \Rightarrow trans$

REPL : $\vdash \iota_1 \times [\tau_2] \to [\tau_1 \times \tau_2] \Rightarrow distl$

REPR : $\vdash [\tau_1] \times \iota_2 \to [\tau_1 \times \tau_2] \Rightarrow distr$

RFLX : $\vdash \tau \lesssim \tau \Rightarrow id$

TRNS : $$\frac{\vdash \tau_1 \lesssim \tau_2 \Rightarrow f \qquad \vdash \tau_2 \lesssim \tau_3 \Rightarrow g}{\vdash \tau_1 \lesssim \tau_3 \Rightarrow g \circ f}$$

LIST : $$\frac{\vdash \tau_1 \lesssim \tau_2 \Rightarrow f}{\vdash [\tau_1] \lesssim [\tau_2] \Rightarrow \alpha f}$$

PROD : $$\frac{\vdash \tau_1 \lesssim \tau_2 \Rightarrow f \qquad \vdash \tau_3 \lesssim \tau_4 \Rightarrow g}{\vdash \tau_1 \times \tau_3 \lesssim \tau_2 \times \tau_4 \Rightarrow \{f.1, g.2\}}$$

FUN : $$\frac{\vdash \tau_1 \lesssim \tau_2 \Rightarrow f \qquad \vdash \tau_3 \lesssim \tau_4 \Rightarrow g}{\vdash \tau_2 \to \tau_3 \lesssim \tau_1 \to \tau_4 \Rightarrow \lambda h.\, g \circ h \circ f}$$

[(3, [1,2]),
([4,5], 6)]

⟶ ?

$$[(3, [1,2]),\ ([4,5], 6)]$$

$$\longrightarrow\!\!\!\!\!\rightarrow\ [\,distl\ (3, [1,2]),\ distr\ ([4,5], 6)]$$

$$= \begin{bmatrix} [(3,1), (3,2)], \\ [(4,6), (5,6)] \end{bmatrix}$$

$$: [[int \times int]]$$

$$A \vdash e \Rightarrow e' : \tau$$

$$A \vdash nil_\tau \Rightarrow nil_\tau : [\tau]$$

$$\frac{A \vdash e_1 \Rightarrow e_1' : \tau \qquad A \vdash e_2 \Rightarrow e_2' : [\tau]}{A \vdash e_1 :: e_2 \Rightarrow e_1' :: e_2' : [\tau]}$$

$$\vdots$$

$$\frac{A \vdash e \Rightarrow e' : \tau_1 \qquad \vdash \tau_1 \leq \tau_2 \Rightarrow f}{A \vdash e \Rightarrow f\, e' : \tau_2}$$

Insertion of coercions is made explicit by Typing Rules

Properties of $\leq$

- RFLX
- TRNS
- ANTISYMMETRIC

subtyping rules

proved in paper

- Coherent :

$$\frac{\vdash \tau_1 \leq \tau_2 \Rightarrow f \qquad \vdash \tau_1 \leq \tau_2 \Rightarrow g}{f \equiv g \text{ (extensionally)}}$$

f may differ syntactically from g.

eg) $\vdash [int \times int] \to int \leq [[int] \times [int]] \to [int]$

$\Rightarrow (\lambda g.\, g \circ (\alpha\ trans)) \circ \alpha$

$\equiv \alpha \circ (\lambda g.\, g \circ trans)$

A SEMANTICS FOR SHAPE - C. BARRY JAY '95

DATA POLYMORPHISM:

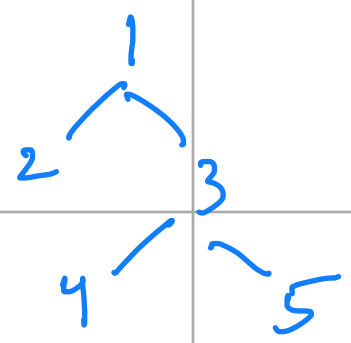
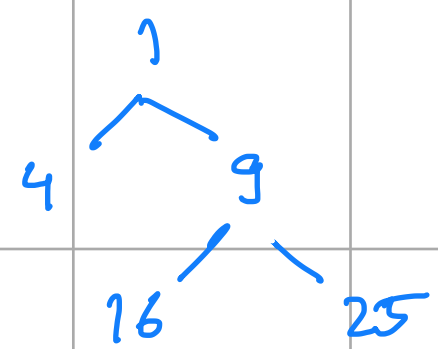
$map_\tau : (\alpha \to \beta) \to \alpha\ list \to \beta\ list$

map_τ square [1, 2, 3] = [1, 4, 9]

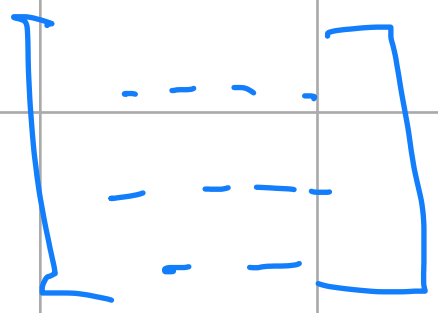
map_τ itoa [1, 2, 3] = ['a', 'b', 'c']

SHAPE POLYMORPHISM:

$map_\sigma : (int \to int) \to int^{\sigma_1} \to int^{\sigma_2}$

map_σ square  = 

map_σ square [1,2,3] = [1,4,9]

matrix Mul [...][⋮] = 

- idea: Shape of the result does not always depend on input data.

common in data parallel computations.

★ Can use input shapes for load balancing on multiple CPUs!

Paper shows that under mild assumptions, the existence of lists is enough to establish the existence of all the other inductive types, such as trees.

Regular, Shape Polymorphic, Parallel Arrays in Haskell

— Keller, Chakravarty, Leshchinskiy, Jones & Lippmeier, 2010

— Repa: A Haskell library providing a regular Array Datatype with parallel aggregate operations.

Some examples:

```
extent :: Array sh e → sh
```

shape

```
sum :: (Shape sh, Elt e, Num e)
    ⇒ Array (sh :. Int) e → Array sh e
```

snoc operator

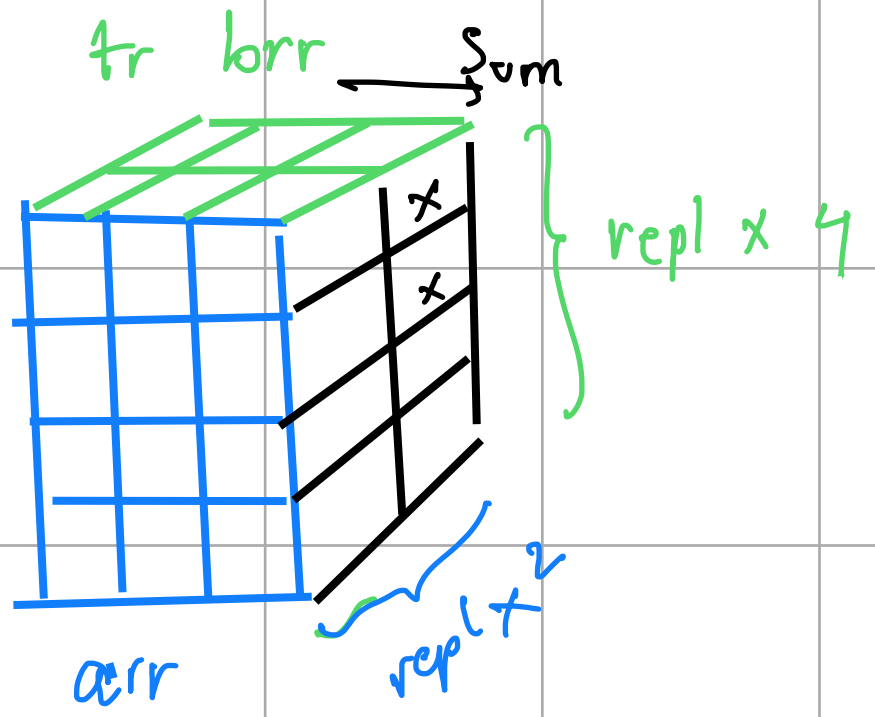
```
zipWith :: (Shape sh, Elt e1, Elt e2, Elt e3)
        => (e1 -> e2 -> e3)
        -> Array sh e1 -> Array sh e2
        -> Array sh e3
```

```
transpose2D :: Elt e => Array DIM2 e ->
               Array DIM2 e
```

```
mmMult :: (Num e, Elt e)
       => Array DIM2 e -> Array DIM2 e
       -> Array DIM2 e
mmMult arr brr
  = sum (zipWith (*) arrRepl brrRepl)
  where
    trr = transpose2D brr
    arrRepl = replicate (Z :. All :. colsB :. All) arr
    brrRepl = replicate (Z :. rowsA :. All :. All) trr
    (Z :. colsA :. rowsA) = extent arr
    (Z :. colsB :. rowsB) = extent brr
```

4 x 3
arr

3 x 2
brr



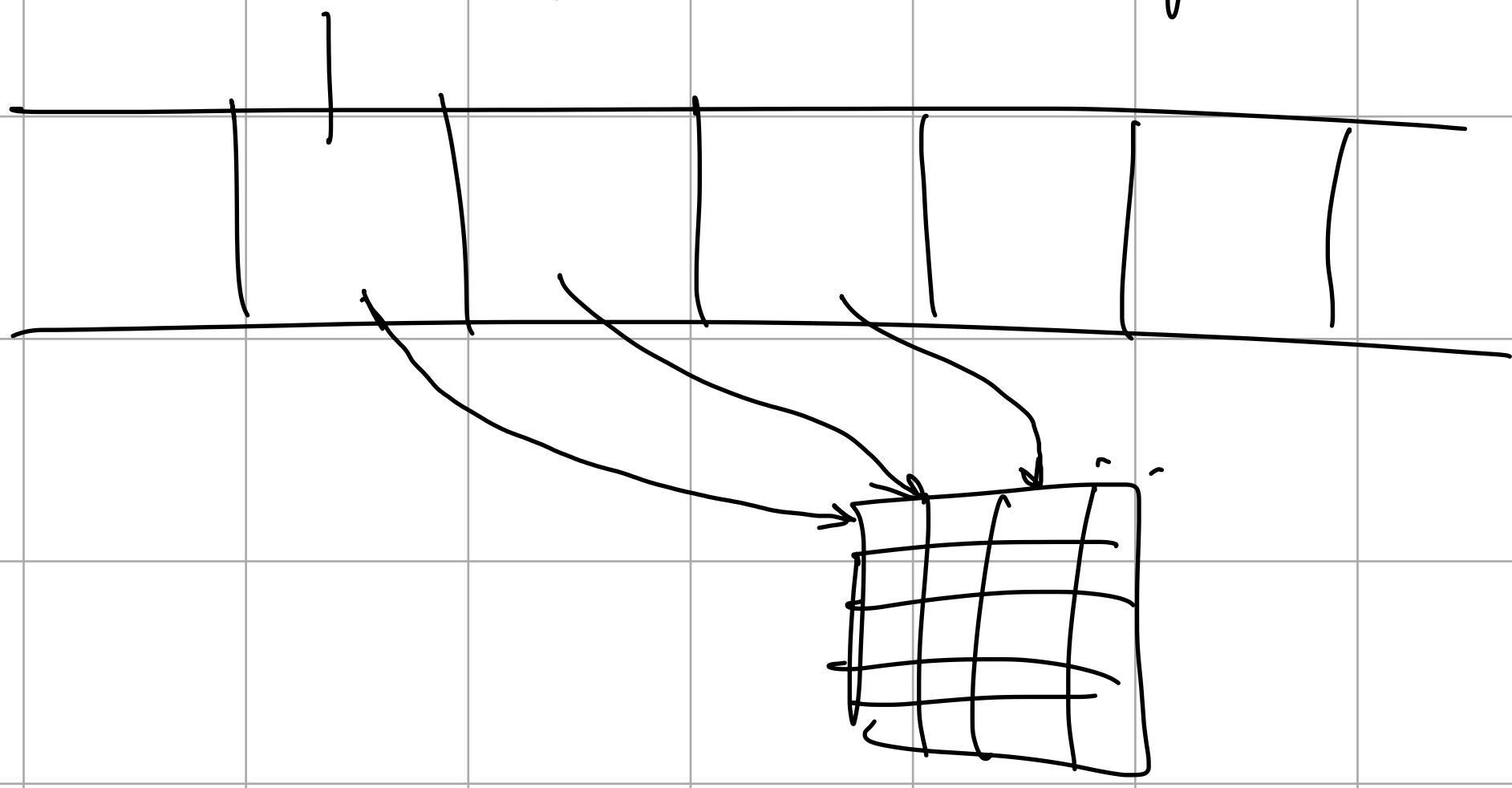
Wouldn't transposing big arrays be expensive?

idea:

```
data DArray sh e = DArray sh (sh -> e)
```

indexing function

f . g . h . idx ..... eval lazily



WHAT DID THEY MISS?

Type system for implicit scaling
- Thatte '91

- Loses co-herence when combined with implicit parametric polymorphism of HM.

Shape Polymorphism
- Cockett, Jay '94
- REPA - KELLER ET.AL

- severely restricted computation, output shape HAS to depend ONLY on the input shape

Can't have L, S

- no implicit lifting

Under parametric polymorphism,
hd is usually a polymorphic function
whose type is of the form:

$$[\tau] \to \tau$$

type assumptions
↓

$A \vdash hd \Rightarrow hd : [[int]] \to [int]$,

since $A(hd) = [[int]] \to [int]$

$A \vdash hd \Rightarrow^{\kappa} hd : [[int]] \to [int]$

since $A(hd) = [int] \to int \leq [[int]] \to [int]$

REMORA

– SLEPAK, SHIVERS, MANOLIOS 2014

- higher order
- rank polymorphic
  - functions accept arguments of arbitrarily high rank.
- static type system that can infer shape of runtime data

How?

By using a restricted form of dependent typing

SAMPLE REMORA TERMS

```
(array () 5)
```

ARRAY

5

atoms

```
(array (2 2) 1 2 3 4)
```

ARRAY

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}_{2\times 2}$$

```
(+ [[1 2 3]
   [4 5 6]] [1 2])
```

FUNCTION APPLICATION

$$\left(+ \;\; \underset{2\times 3}{\begin{bmatrix}[1\ 2\ 3]\\ [4\ 5\ 6]\end{bmatrix}} \;\; [1\ 2]_2\right)$$

ranks $[\,]$ $[2\ 3]$ $[2\,]$

$+ : \cdot, \cdot \to \cdot$ principal frame

scaling is part of f-app

function app cells

$$\left[\begin{bmatrix}(+\ 1\ 1) & (+\ 2\ 1) & (+\ 3\ 1)\end{bmatrix}\ \begin{bmatrix}(+\ 4\ 2) & (+\ 5\ 2) & (+\ 6\ 2)\end{bmatrix}\right]$$

FRAME

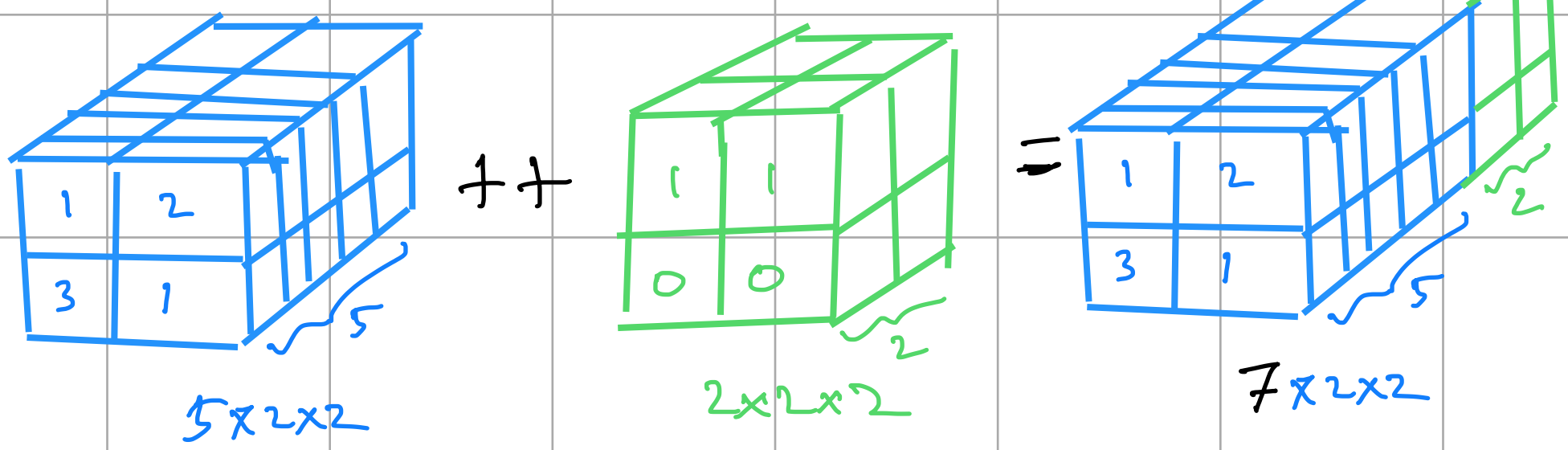
ARRAYS

$$\begin{bmatrix}[2\ 3\ 4]\\ [6\ 7\ 8]\end{bmatrix}$$

ARRAY

```
length : (→
          (Π ((s Shape) (d Dim))
             (∀ (t Atom)
                ((A t (++ (shape d) s)))
                (A Int (shape)))))

append : (→ (... (... ((A t (++ shape m) s))
                       (A t (++ shape n) s)))
            (A t (++ shape (+ m n)) s)))
```



Results:

Soundness Theorem: well-typed programs will not suffer from shape-mismatch errors.

Sample program in typed Remora

```
((array [] (foldl) (A (→ ((A R (Shape []))
                          (A R (Shape [])))
                         (A R (Shape [])))
                      (Shape []))
                      (A R (Shape d])))
                   (Shape []))

(array () (+) (A (→ ((A R (Shape []))
                     (A R (Shape [])))
                    (A R (Shape []))))
              (Shape []))

(array () (0) (Shape []))
              (A R (Shape [])))

(array (5) (12345) R))
```

writing types is tedious!

Type Inference in Implicitly typed Remora

An untyped Remora code:

```
( λ ( x )
    ( + x
      [[1 0]
       [0 1]] ))
```

Can't use Hindley-Milner due to lack of principal types!

ideas:

- Rank annotations
- Use bi-directional type checking
- A theory solver for the theory of type indices.

Bidirectional type checking

- Uses the Pfenning recipe
- Synthesize types of elim forms
- Check types of intro forms

rank annotation

```
(λ ([x 2])
  (+ x
     [[1 0]
      [0 1]]))
```

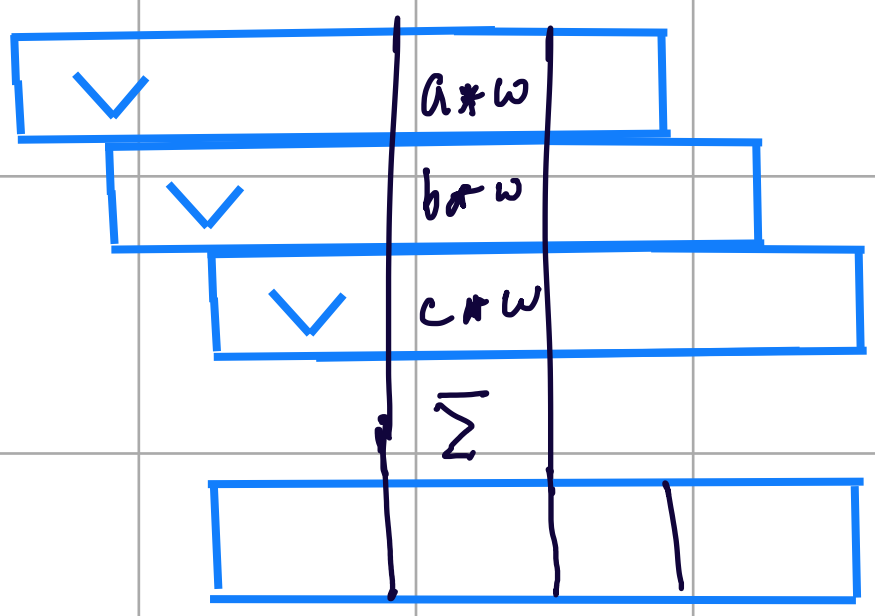
infer t : [Int 2 2] → [Int 2 2] → [Int 2 2]

check x : [Int a b]

$\underbrace{\phi}_{\text{solver}} = \{[a\ b] = [2\ 2]\}$

```
(define (mean [xs 1])
  (/ (reduce + 0 xs)
     (length xs)))
```

```
(define (vector-convolve [v 1] [w 1])
  (reduce + 0
          (* (rotate v (iota (shape w)))
             w)))
```



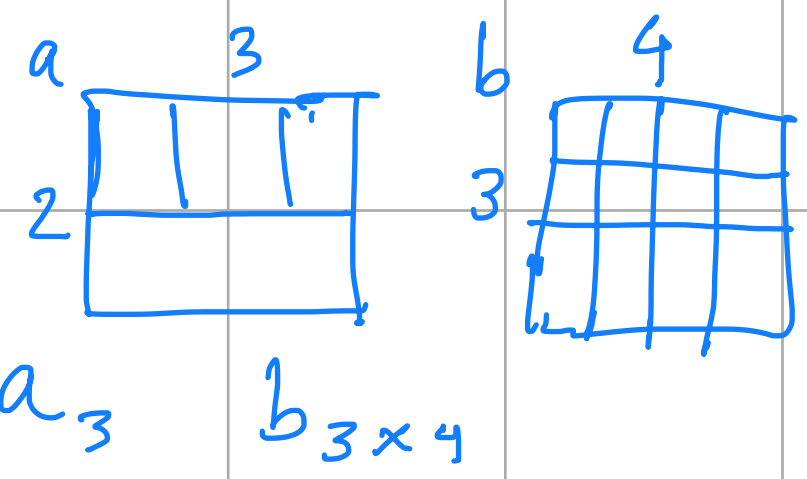
```
(define (m*m [a 2] [b 2])
  (~ (0 0 2) reduce +
     0
     (~ (1 2) * a
        b )))
```

~ (1 2) * a b

↓

```
(λ ([a :] [b 2])
  (* a b))
```

↓



(*). a_3 $b_{3\times 4}$

Remora Pros

- Expressive
- Higher order

```
(foldl [sum square]
       [0   0]
       [v   w])
```

- Rank Polymorphic static type system
- With a meta theory in place : Soundness Theorem
- And a type-inference algorithm

Cons :

- We don't have a compiler (YET)

- Constraint annotation is not supported

```
(λ ([x 1] [y 1])
   (+ (append x y)
      [1 2 3 4 5]))
```

constraint: $|x| + |y| = 5$

- Type inference depends on solving string equations, whose complexity is in PSPACE.
  Makanin first showed in 1977 that the problem is decidable

Several string solvers exist today, but aren't complete, owing to very high bounds on minimal length of a solution.

eg.

$$xcyczvycya = yacwazvbux$$

is unsolved by CVC4, Norn, Z3str2 and Z3str3

Pete and I recently submitted work on SeqSolve that solves the above in < 1 sec.

Conclusion

APL was a great idea :

a solution to the von-neumann bottleneck, has no iteration / recursion. Has aggregate operations.

Is it relevant today ?

- Need : Machine learning, data processing.

Software used today : PyTorch, NumPy ..

But these are just libraries, with no language support

- Hardware: Parallelism is not an opportunity. It's a problem we can no longer ignore.

  Today, it is easier to build a parallel computer than to program one.

  GPUs give us the ability to run SIMD programs. Software is the bottleneck.

  SIMD + λ = MIMD

  Remora code:

  ([+ *] 3 4) ⇒ [7 12]

  – taken from Olin's talk

“Everywhere I have presented Remora, I have had the same conversation with 5-6 different people asking if they can have the type system of Remora in X, where X is their choice of programming language”.

-Olin