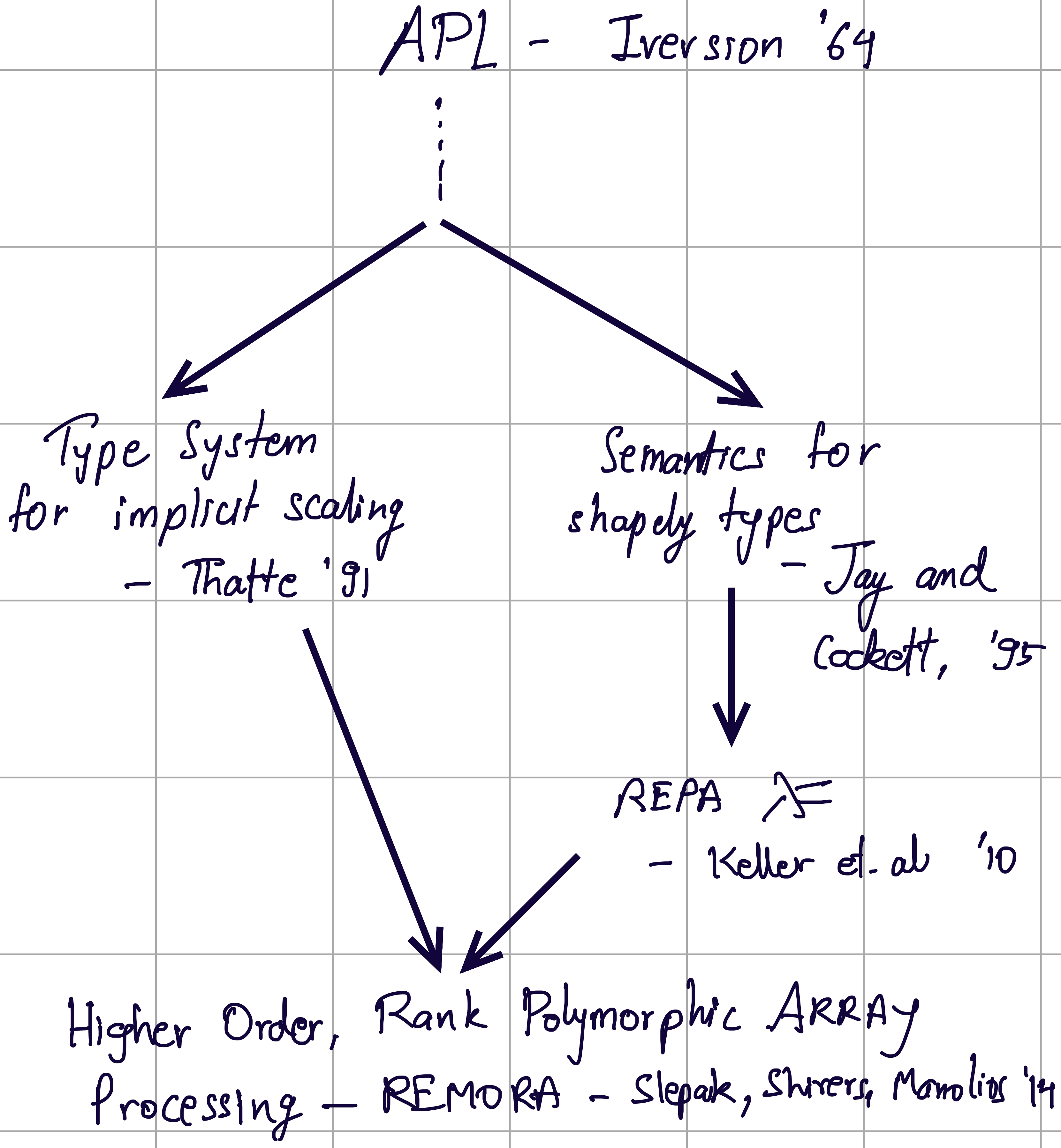
History of Rank Polymorphism in Array Programming Languages



APL - Iverson '64

Type System for implicit scaling - Thatte '91

Semantics for shapely types - Jay and Cockett, '95

REPA - Keller et. al '10

Higher Order, Rank Polymorphic ARRAY Processing - REMORA - Slepak, Shivers, Manolios '14

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:

- ⍳5

  1 2 3 4 5

- R ← ⍳5

  R ← ⍳5

- +/R

  15

- 2↓R

  3 4 5

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

- 1.2 + 2.3

  3.5

*1

2.718281828

⍟ 2.718281826

0.9999999998

- 3 × 4 + 5

  ←

  27

- (3 × 4) + 5

  17

- 1 1 0 1 ∧ 1 0 1 1

1 0 0 1

- 1 1 0 1 ∧ 0

0 0 0 0

- ⍴ ⍳ 5

5

- 3 3 ⍴ ⍳ 5

1 2 3
4 5 1
2 3 4

- (3 3 ⍴ 1) × 3 3 ⍴ ⍳ 9

?

- ⌈ ⍳ 3

3

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

- $M +.\times N$

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

- $M \lceil.+ N$?

- $(\iota 3) \circ.\times (\iota 3)$

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

- $(\iota 3) \circ.= (\iota 3)$?

```
(~R ∈ R ∘.× R) /R ← 1 ↓ ⍳100
```

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

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

$$\downarrow$$

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

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."

APL uses aggregate operations in lieu of iteration or recursion

`A * n` instead of

```
for (i=0; i<100; i++)
  for (j=0; j<100; j++)
    B[i][j] = A[i][j] * n
```

ARRAY PROGRAMMING was huge in the 70s

Iverson was awarded the Turing Award in 1979 for designing APL, and also received a mention in Backus' speech the previous year.

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:

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

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

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

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

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

- Complex semantics :

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

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

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

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

- Expressiveness:

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

ML:

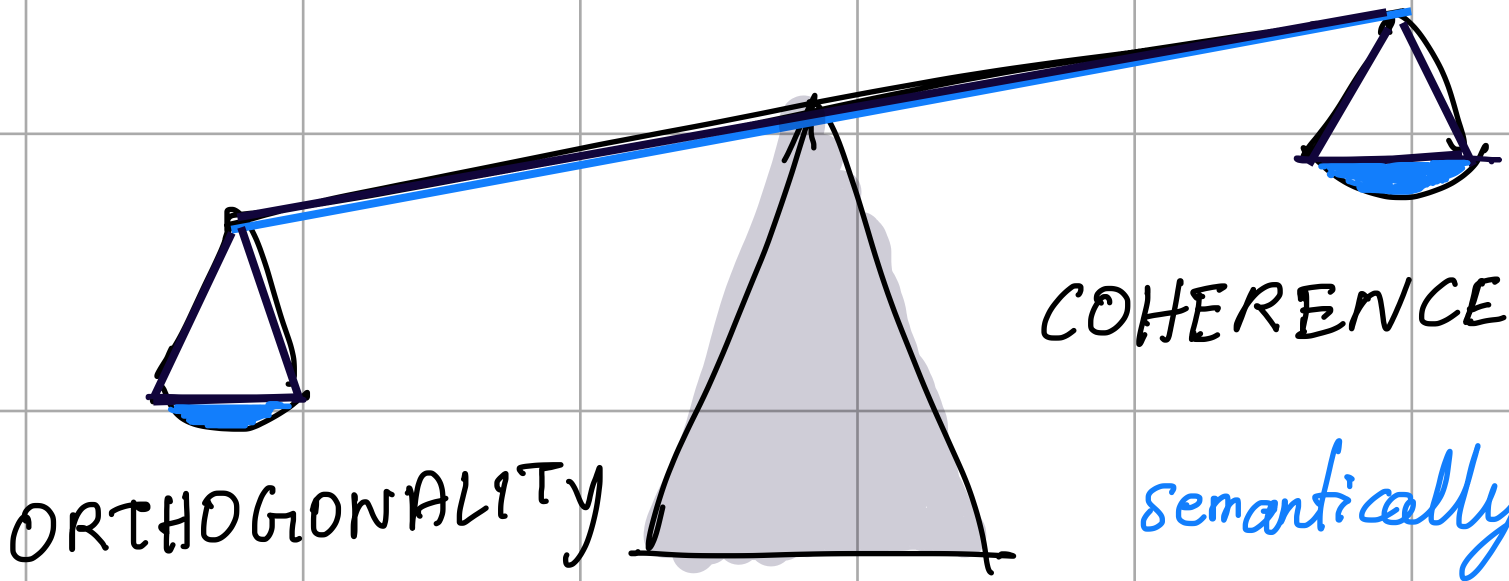
```
map (op+) (distl (a, map (op ×) (distl (b, V))))
```

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 $[e_1, e_2 \dots e_n]$ for

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

τ

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

$$\boxed{e \twoheadrightarrow e'}$$

e is expected to be co-erced to e' by a minimal typing derivation.

eg:

- $\text{square } [1,2,3] \twoheadrightarrow (\alpha \text{ square})\ [1,2,3]$

  $= [1,4,9]$

- $1 + [1,2,3] \twoheadrightarrow (\alpha +)\ (\text{distl } (1, [1,2,3]))$

  $= (\alpha +)\ [(1,1), (1,2), (1,3)))$

  $= [2,3,4]$

- $[1,2,3] + [2,3,4] \twoheadrightarrow (\alpha +)\ (\text{trans } ([1,2,3], [2,3,4])))$

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

  $= [3,5,7]$

$$\boxed{\vdash \tau_1 \leq \tau_2 \Rightarrow f}$$

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

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

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

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

coherence problem

$[\mathrm{int}] \times [\mathrm{int}] \leq [[\mathrm{int}] \times \mathrm{int}] \leq [[\mathrm{int} \times \mathrm{int}]]$

distr . distl

$[\mathrm{int}] \times [\mathrm{int}] \leq [\mathrm{int} \times [\mathrm{int}]] \leq [[\mathrm{int} \times \mathrm{int}]]$

distl . distr

$$\boxed{\vdash \tau_1 \leq \tau_2 \Rightarrow f}$$

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

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

REPL : $\vdash \ell_1 \times [\tau_2] \to [\ell_1 \times \tau_2] \Rightarrow \mathit{distl}$

REPR : $\vdash [\tau_1] \times \ell_2 \to [\tau_1 \times \ell_2] \Rightarrow \mathit{distr}$

$$\boxed{\vdash \tau_1 \leq \tau_2 \Rightarrow f}$$

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

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

REPL : $\vdash l_1 \times [\tau_2] \to [\tau_1 \times \tau_2] \Rightarrow distl$

REPR : $\vdash [\tau_1] \times l_2 \to [\tau_1 \times \tau_2] \Rightarrow distr$

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

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

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

PROD : $$\frac{\vdash \tau_1 \leq \tau_2 \Rightarrow f \qquad \vdash \tau_3 \leq T_4 \Rightarrow g}{\vdash \tau_1 \times \tau_3 \leq \tau_2 \times \tau_4 \Rightarrow \{f.1, g.2\}}$$

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

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

$$\twoheadrightarrow \quad ?$$

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

$$\twoheadrightarrow \begin{aligned}[&\mathrm{distl}\ (3, [1,2]), \\ &\mathrm{distr}\ ([4,5], 6)]\end{aligned}$$

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

$$: [[\mathrm{int} \times \mathrm{int}]]$$

$$\boxed{A \vdash e \Rightarrow e' : \tau}$$

$$A \vdash \mathrm{nil}_\tau \Rightarrow \mathrm{nil}_\tau : [\tau]$$

$$\frac{\begin{array}{l} A \vdash e_1 \Rightarrow e_1' : \tau \\ A \vdash e_2 \Rightarrow e_2' : [\tau] \end{array}}{A \vdash e_1 :: e_2 \Rightarrow e_1' :: e_2' : [\tau]}$$

$$\vdots$$

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

Insertion of coercions is made explicit by Typing Rules

Properties of $\leq$

$$\bullet\ \underbrace{\text{RFLX} \quad \bullet\ \text{TRNS}}_{\text{subtyping rules}} \qquad \bullet\ \underbrace{\text{ANTISYMMETRIC}}_{\text{proved in paper}}$$

- Coherent :

$$\frac{\begin{array}{l} \vdash \tau_1 \leq \tau_2 \Rightarrow f \\ \vdash \tau_1 \leq \tau_2 \Rightarrow g \end{array}}{f \equiv g \ \text{(extensionally)}}$$

f may differ syntactically from g.

eg) $\vdash [\text{int} \times \text{int}] \to \text{int} \leq [[\text{int}] \times [\text{int}]] \to [\text{int}]$

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

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

A SEMANTICS FOR SHAPE - C. BARRY JAY '95

DATA POLYMORPHISM :

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

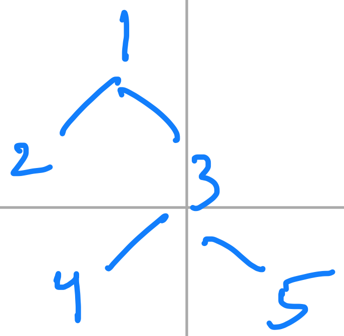
$$map_\tau\ square\ [1,2,3] = [1,4,9]$$

$$map_\tau\ itoa\ [1,2,3] = ['a','b','c']$$

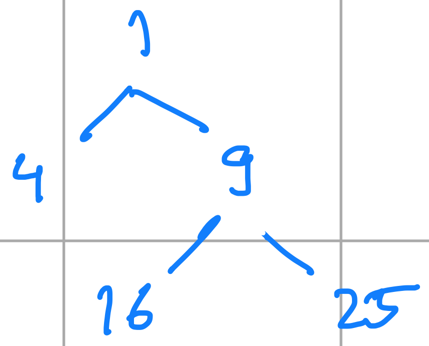
SHAPE POLYMORPHISM:

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

map_{σ} square

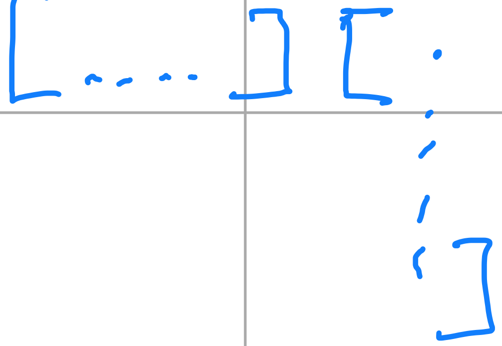


=



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:

shape

`extent :: Array sh e → sh`

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

snoc operator

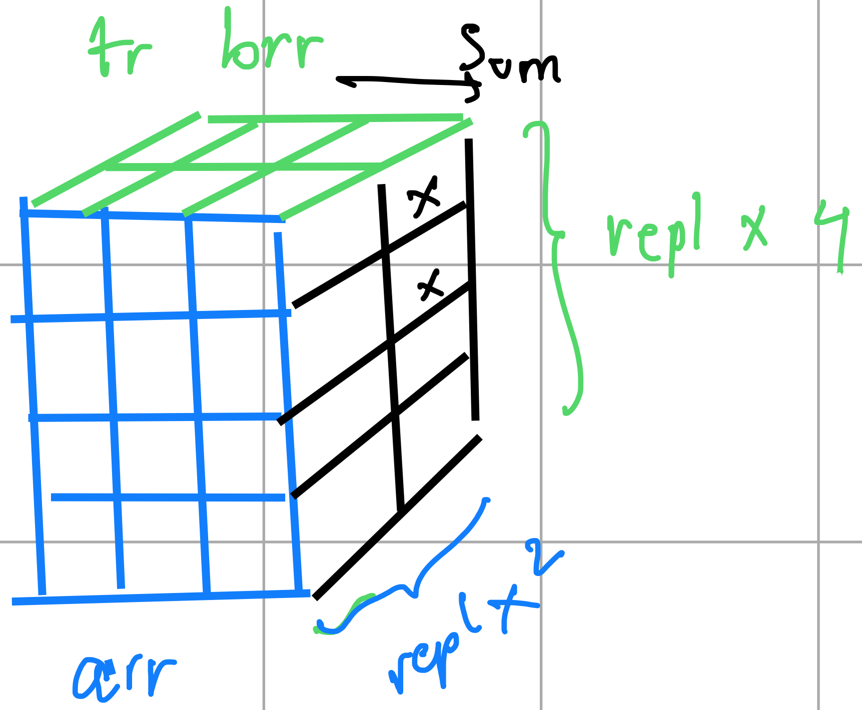
```
zipWith :: (Shape sh, Elt e1, Elt e2, Elt e3)
  ⇒ (e1 → e2 → e3)
   → Array sh e1 → Array sh e2
   → Array sh e3
```

```
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

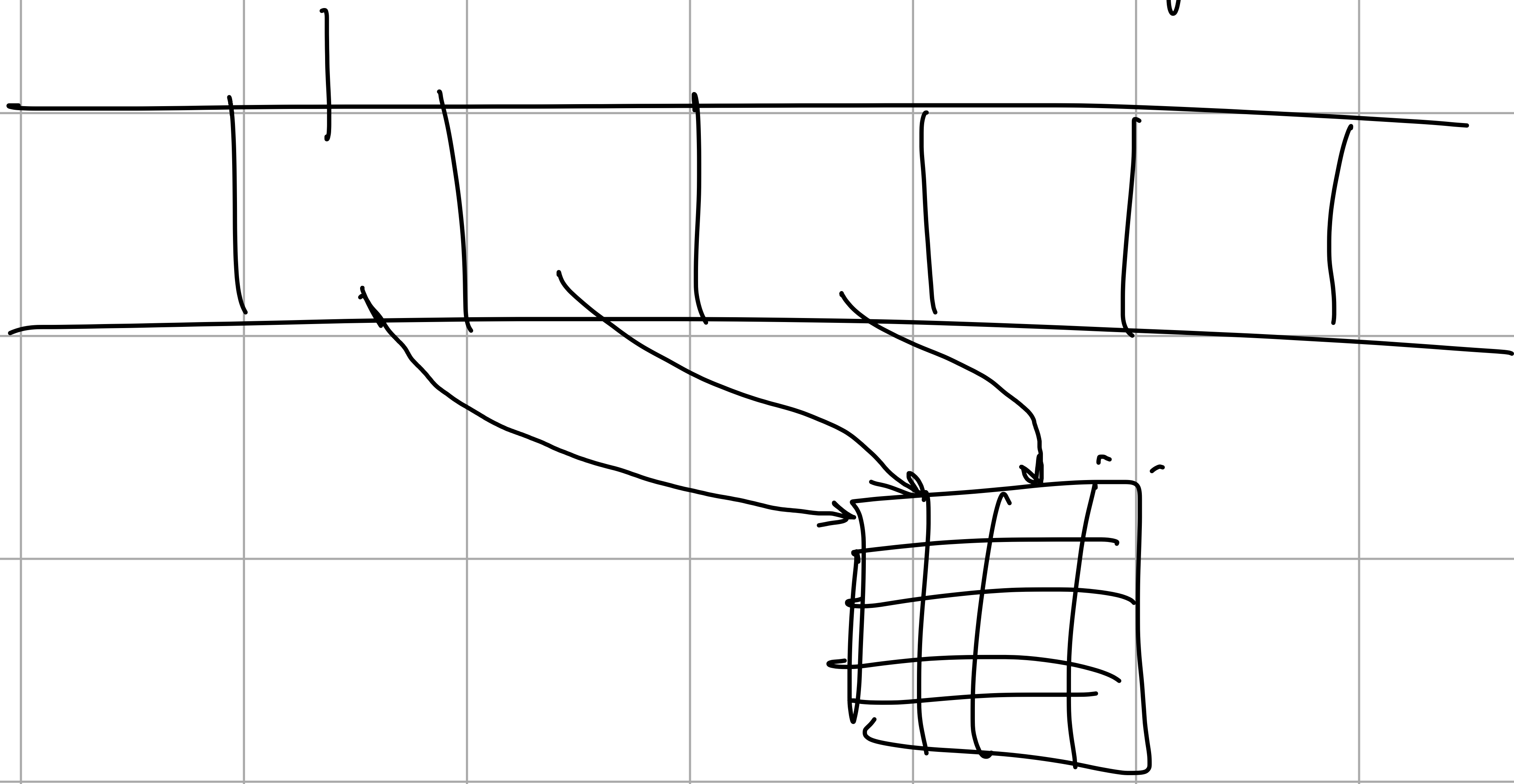


Wouldnt transposing big arrays be expensive?

idea:

$$\text{data DArray sh e = DArray sh } \underbrace{(sh \to e)}_{\text{indexing function}}$$

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



WHAT DID THEY MISS?

Type System for implicit scaling

- Thatte '91

- Loses co-herence when combined with implicity 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

↓

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

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

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

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

atoms

5

(array (2 2) 1 2 3 4)

ARRAY

$$\begin{bmatrix}1 & 2\\ 3 & 4\end{bmatrix}_{2\times2}$$

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

FUNCTION APPLICATION

$$\left(+. \quad \begin{bmatrix} [1 & 2 & 3] \\ [4 & 5 & 6] \end{bmatrix}_{2\times3} \quad [1 \;\; 2]_2 \right)$$

ranks $[\;]$ $\underbrace{[2 \;\; 3]}_{\text{principal frame}}$ $[2]$

$+ : \cdot, \cdot \to \cdot$

scaling is part of f-app

$$\downarrow$$

function app cells

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

FRAME

ARRAYS

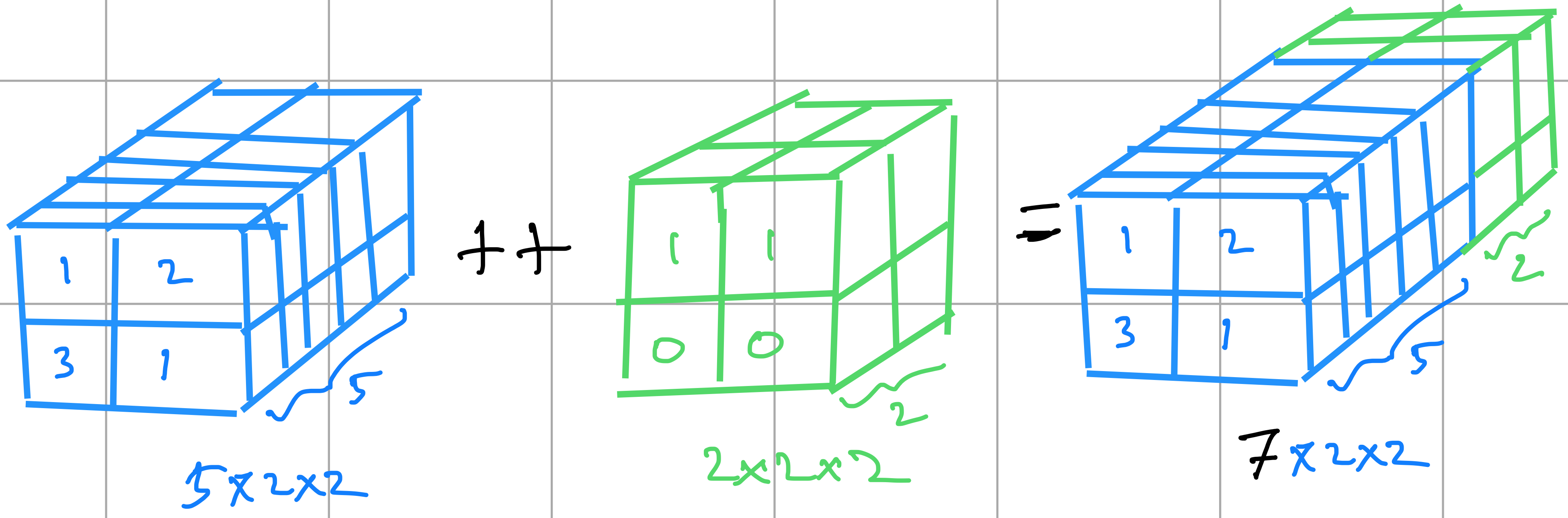
$$\downarrow$$

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

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) (A R (Shape ()))))
 (array (5) (1 2 3 4 5) R))
```

writing types is tedious!

Type Inference in Implicitly typed Remora

An untyped Remora code :

rank annotation

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

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

ideas:

- 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

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

Infer

$+ : [\mathrm{Int}\ 2\ 2] \to [\mathrm{Int}\ 2\ 2] \to [\mathrm{Int}\ 2\ 2]$

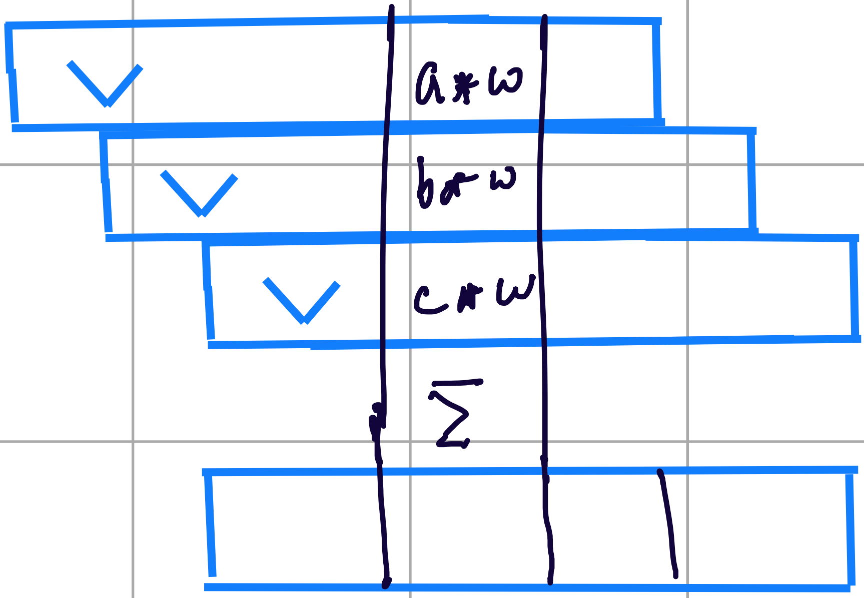
Check

$x : [\mathrm{Int}\ a\ b]$

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

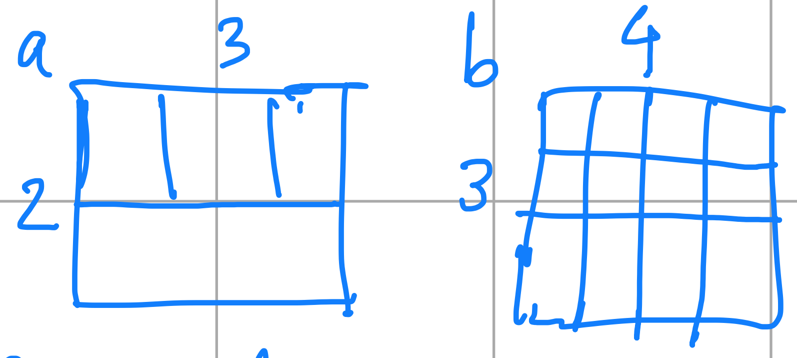
```
(define (mean [xs 1])
   (/ (reduce + 0 xs)
      (length xs)))
```

```
(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 1] [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 metatheory 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.

$$x\,c\,y\,c\,z\,v\,y\,c\,y\,a = y\,a\,c\,w\,a\,z\,v\,b\,u\,x$$

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:

$([+ \; *] \; 3 \; 4) \Rightarrow [7 \; 12]$

– 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