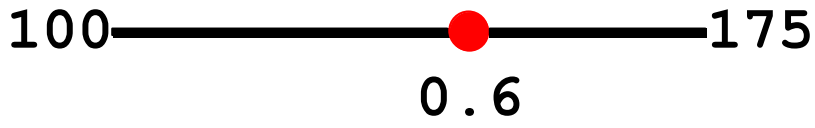
Rank Polymorphism Viewed as a Constraint Problem

Justin Slepak
jrslepak@ccs.neu.edu

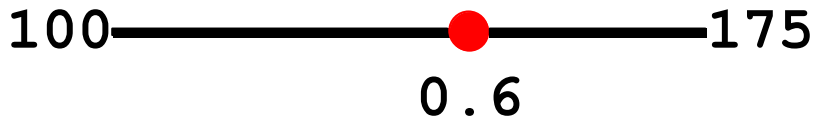
Panagiotis Manolios
pete@ccs.neu.edu

Olin Shivers
shivers@ccs.neu.edu

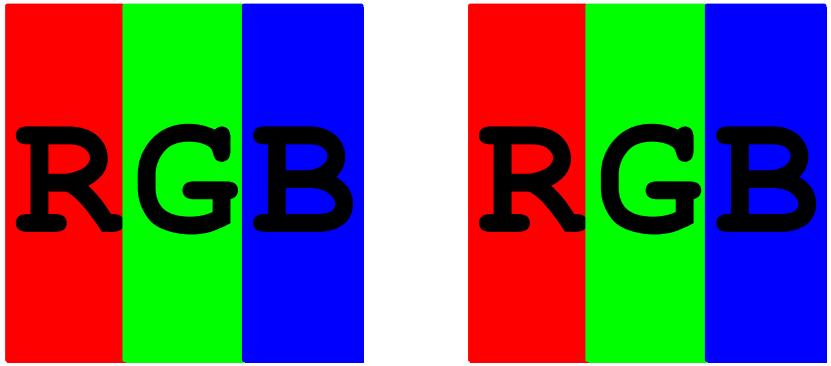
Northeastern University



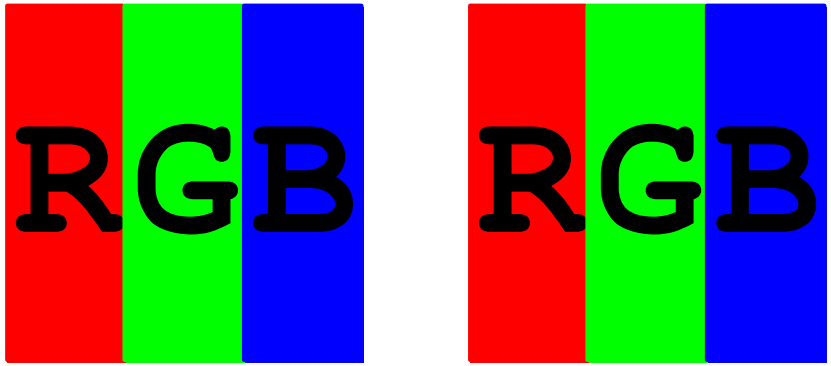

```
(λ [(lo 0) (hi 0) (α 0)]
  (+ (* hi α) (* lo (- 1 α))))
```



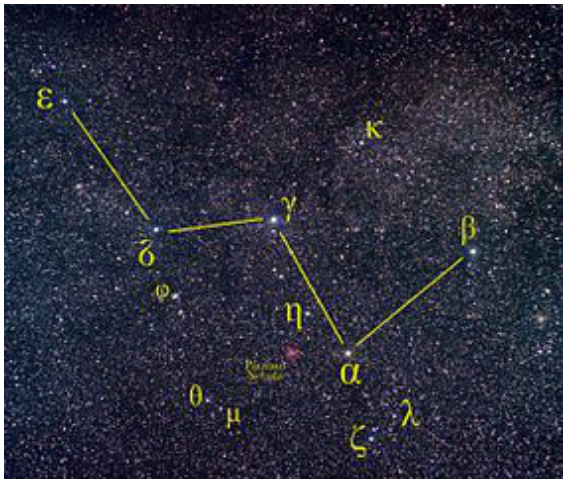

```
((λ [(lo 0) (hi 0) (α 0)]
   (+ (* hi α) (* lo (- 1 α))))
 100
 175
 0.6)
```




```
(λ [(lo 0) (hi 0) (α 0)]
  (+ (* hi α) (* lo (- 1 α))))
```

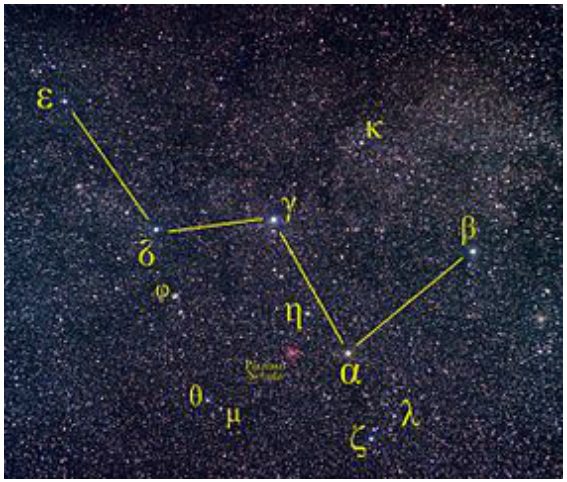



```
((λ [(lo 0) (hi 0) (α 0)]
   (+ (* hi α) (* lo (- 1 α))))
 rgb1 ; 3 channels
 rgb2 ; 3 channels
 0.6) ; scalar
```



Credit: Wikimedia user Sadalsuud

```
(λ [(lo 0) (hi 0) (α 0)]
  (+ (* hi α) (* lo (- 1 α))))
```



Credit: Wikimedia user Sadalsuud

```
((λ [(lo 0) (hi 0) (α 0)]
   (+ (* hi α) (* lo (- 1 α))))
 sky         ; row × col × chan
 labels      ; row × col × chan
 img-mask)   ; row × col × chan
```

```
(λ [(lo 0) (hi 0) (α 0)]
  (+ (* hi α) (* lo (- 1 α))))
```

```
((λ [(lo 0) (hi 0) (α 0)]
   (+ (* hi α) (* lo (- 1 α))))
 film       ; time × row × col × chan
 audience   ; time × row × col × chan
 vid-mask)  ; time × row × col × chan
```

Credit: Wikimedia user Thetawave

```
(λ [(lo 0) (hi 0) (α 0)]
  (+ (* hi α) (* lo (- 1 α))))
```

Credit: Wikimedia user Thetawave

```
((λ [(lo 0) (hi 0) (α 0)]
   (+ (* hi α) (* lo (- 1 α))))
 scene1     ; time × row × col × chan
 scene2     ; time × row × col × chan
 [0.0 ... 1.0]) ; time
```

```
(λ [(lo 0) (hi 0) (α 0)]
  (+ (* hi α) (* lo (- 1 α))))
```

```
(λ [(lo 0) (hi 0) (α 0)]
  (+ (* hi α) (* lo (- 1 α))))
```

Polymorphic in dimensionality

Rank Polymorphic Programming Model

$$\begin{bmatrix} 1 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 \end{bmatrix}_{3,7}$$

$$\begin{bmatrix} 1 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 \end{bmatrix}_{3,7}$$

$$\left[\begin{array}{cc} 0 & 1 \\ 2 & 3 \\ \hline 4 & 5 \\ 6 & 7 \end{array}\right]_{2,2,2}$$

$$\begin{bmatrix} 1 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 \end{bmatrix}_{3,7}$$

$$\left[\begin{array}{cc} 0 & 1 \\ 2 & 3 \\ \hline 4 & 5 \\ 6 & 7 \end{array}\right]_{2,2,2}$$

$$[0].$$

$$\begin{bmatrix} 1 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 \end{bmatrix}_{3,7}$$

atoms: non-aggregate elements

$$\left[\begin{array}{cc} 0 & 1 \\ 2 & 3 \\ \hline 4 & 5 \\ 6 & 7 \end{array}\right]_{2,2,2}$$

$$[0].$$

$$\begin{bmatrix} 1 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 \end{bmatrix}_{3,7}$$

atoms: non-aggregate elements

$$\left[\begin{array}{cc} 0 & 1 \\ 2 & 3 \\ \hline 4 & 5 \\ 6 & 7 \end{array}\right]_{2,2,2}$$

shape: size in each dimension

$$[0].$$

$$\begin{bmatrix} 1 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 \end{bmatrix}_{3,7} \quad 2$$

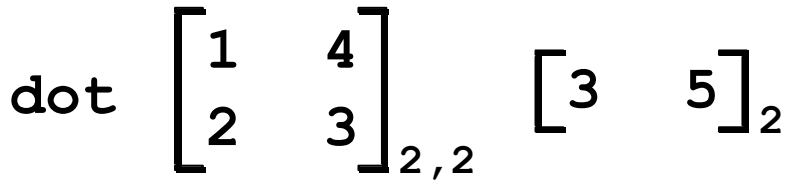
atoms: non-aggregate elements

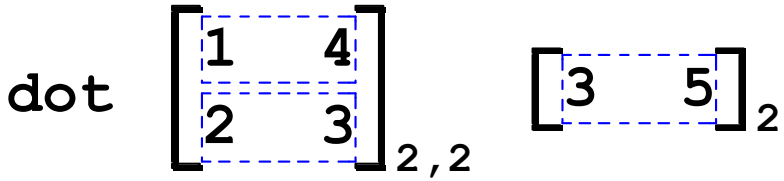
$$\left[\begin{array}{cc} 0 & 1 \\ 2 & 3 \\ \hline 4 & 5 \\ 6 & 7 \end{array}\right]_{2,2,2} \quad 3$$

shape: size in each dimension

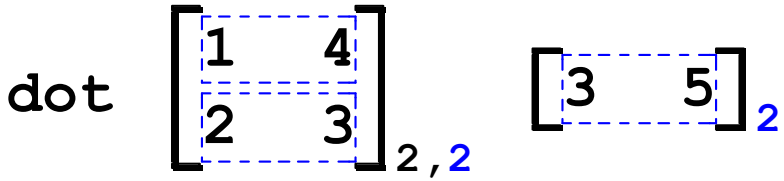
$$[0]. \quad 0$$

rank: length of shape
i.e., number of dimensions

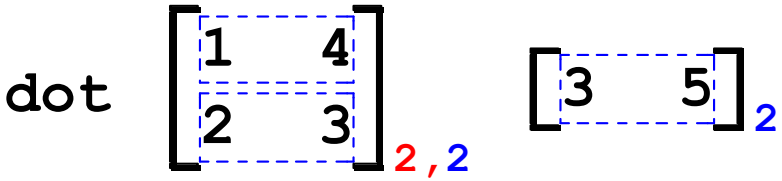

dot
1 4
2 3
2,2
3 5
2



cell: basic unit function operates on

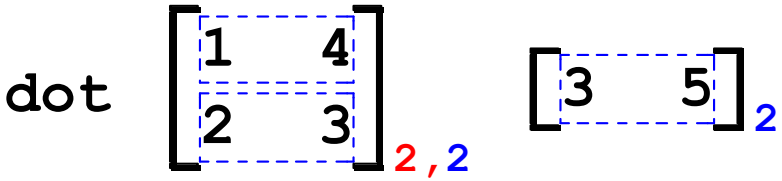


cell: basic unit function operates on



cell: basic unit function operates on

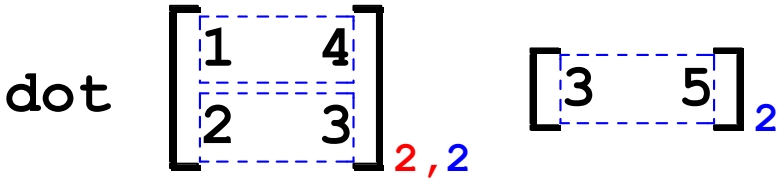
frame: structure around cells



cell: basic unit function operates on

frame: structure around cells

function applied to each cell



cell: basic unit function operates on

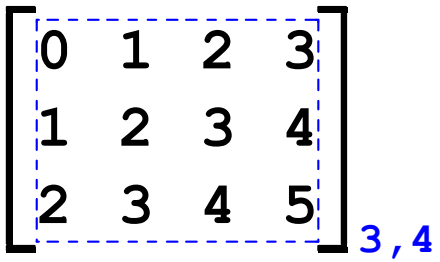
frame: structure around cells

function applied to each cell

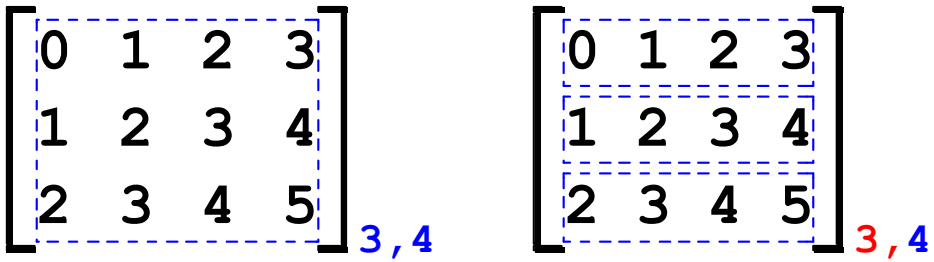
results reassembled in frame

Rank n array can split n+1 ways

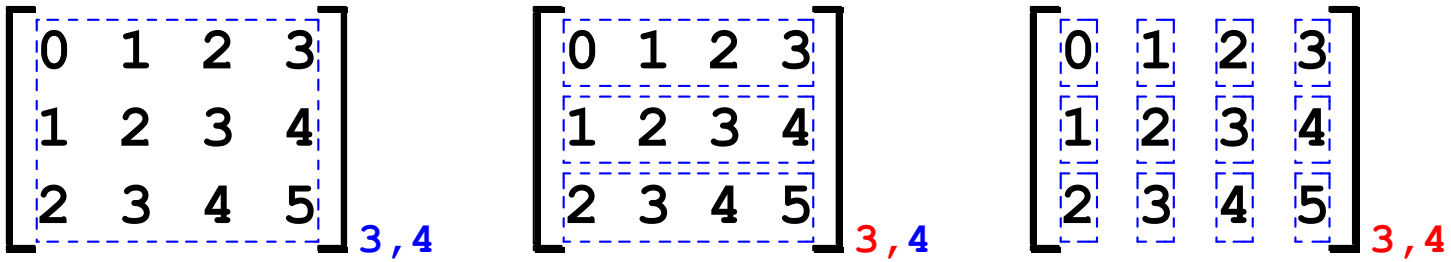
Rank n array can split n+1 ways

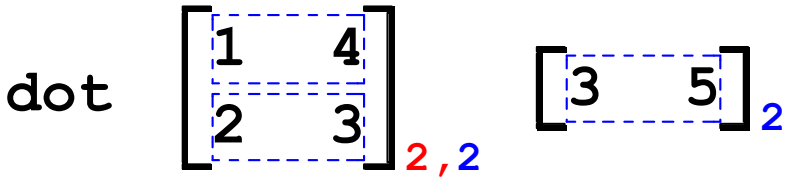


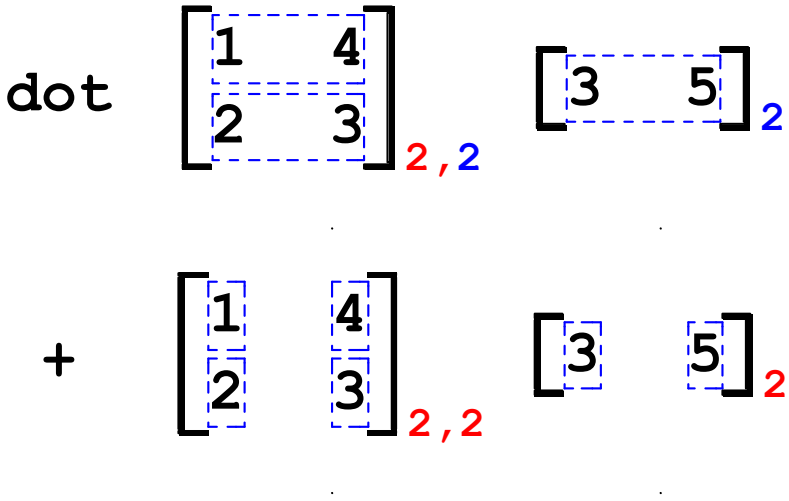
Rank n array can split n+1 ways

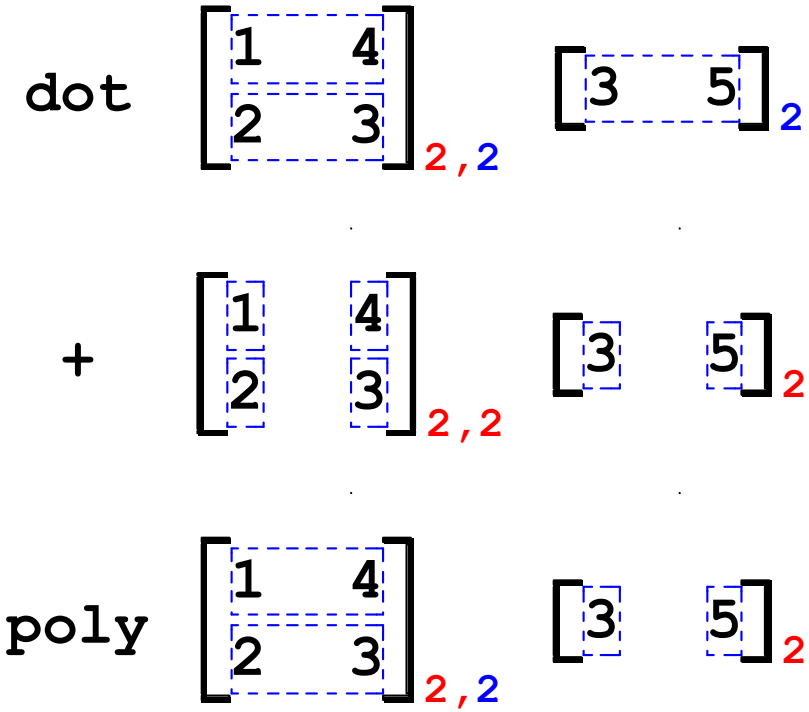


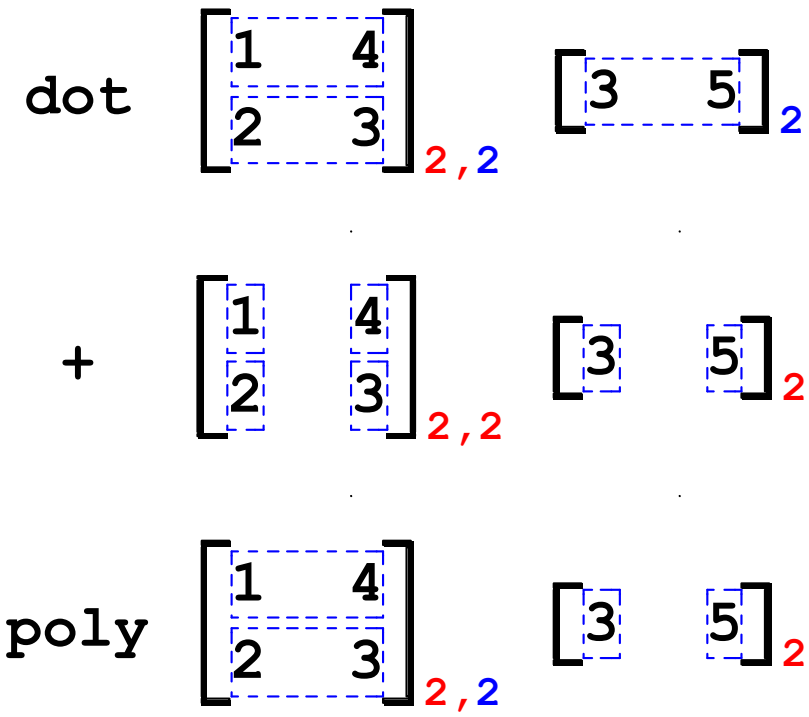
Rank n array can split n+1 ways



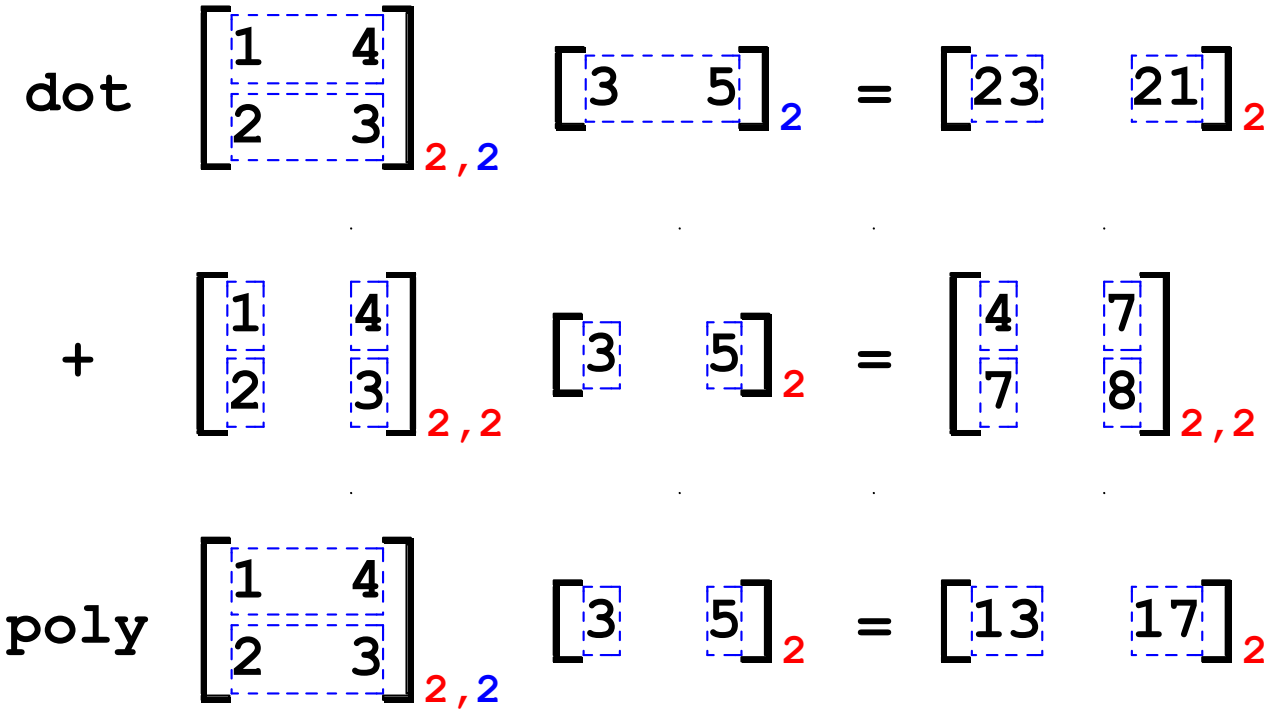

dot
1 4
2 3
2,2
3 5
2


dot
1 4
2 3
2,2
3 5
2
+
1 4
2 3
2,2
3 5
2


dot
1 4
2 3
2,2
3 5
2
+
1 4
2 3
2,2
3 5
2
poly
1 4
2 3
2,2
3 5
2



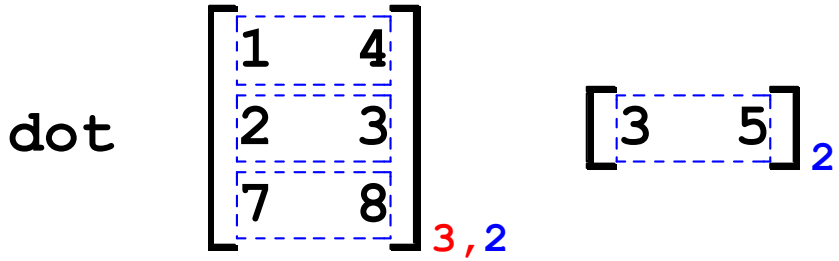
Principal frame: maximum under prefix ordering



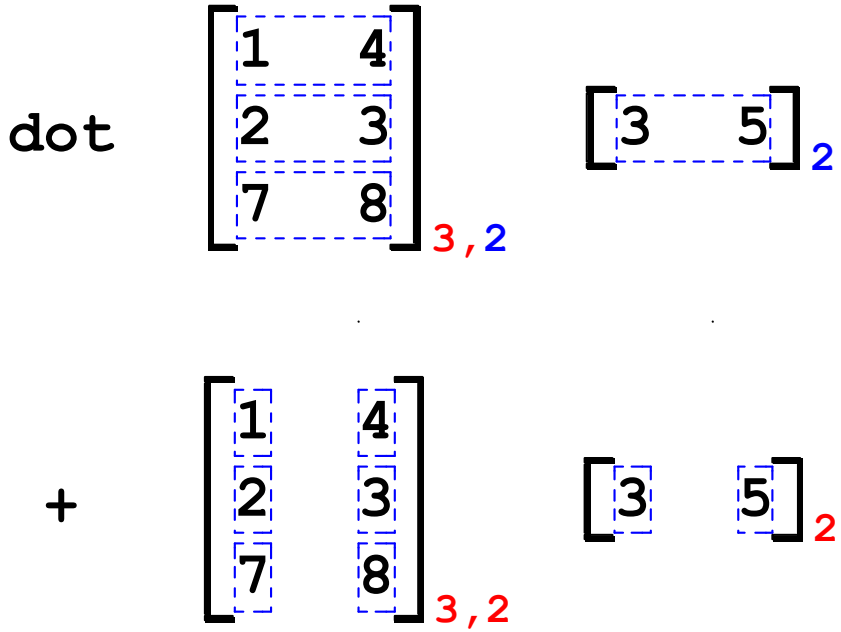
Principal frame: maximum under prefix ordering

Trying different shapes

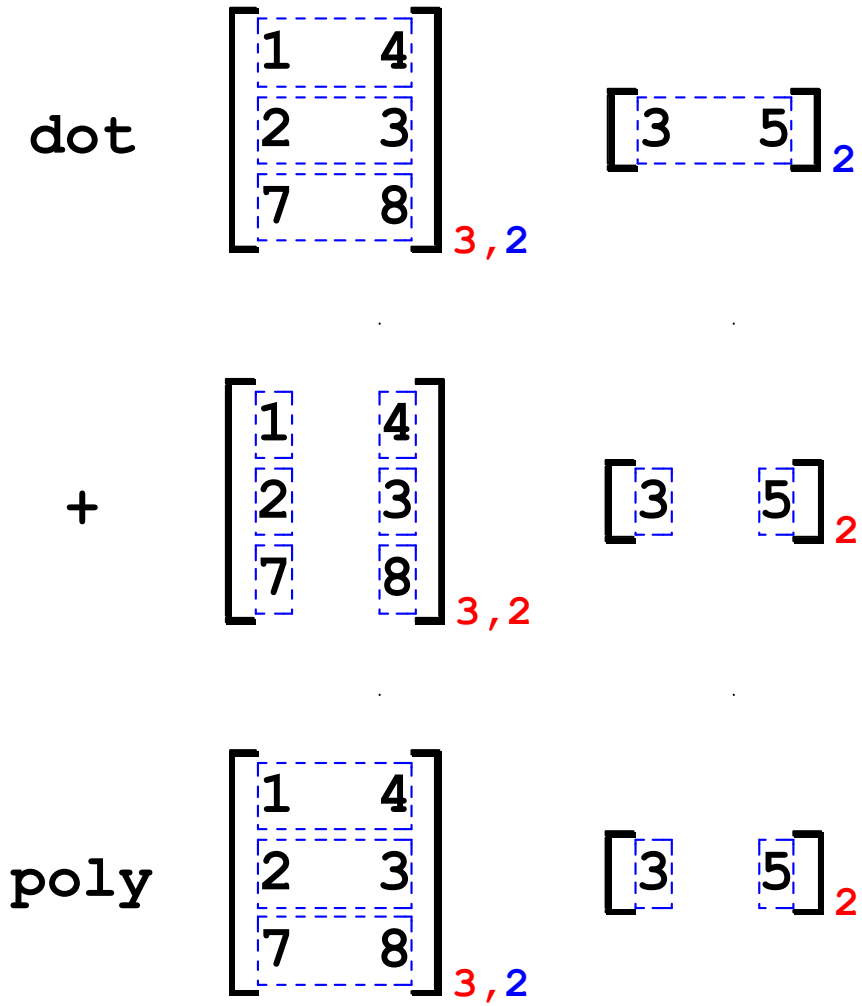
Trying different shapes



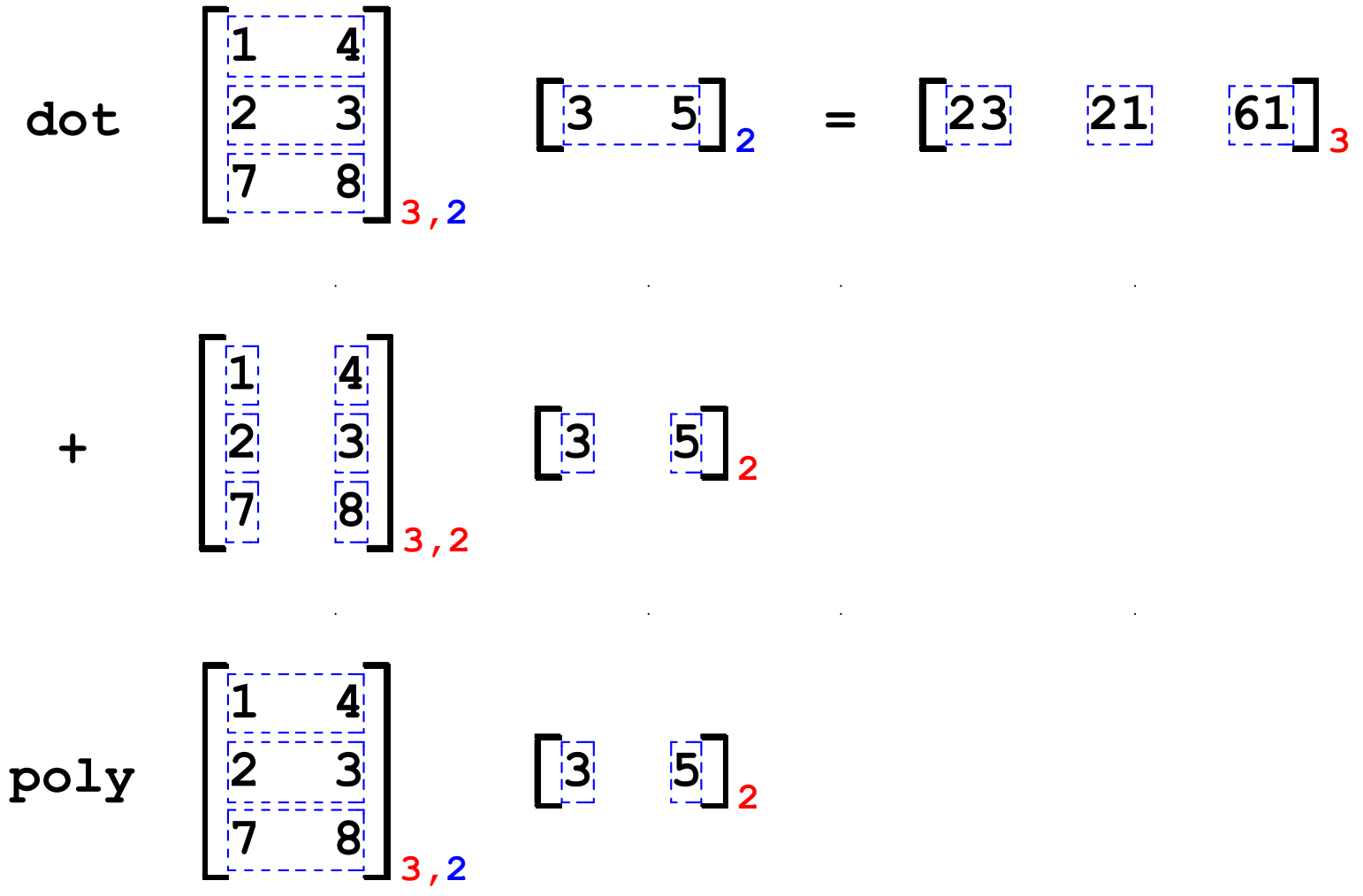
Trying different shapes



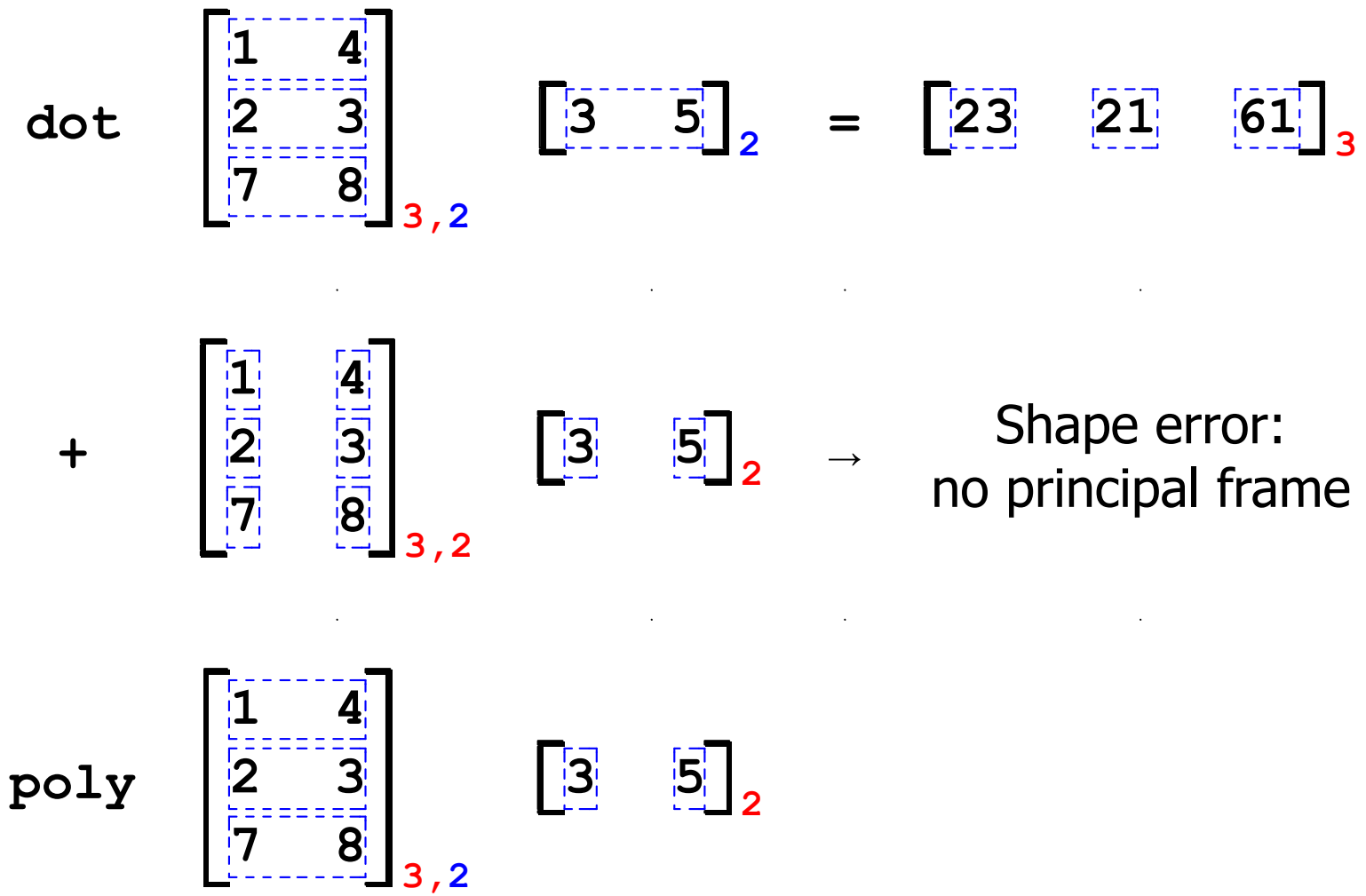
Trying different shapes



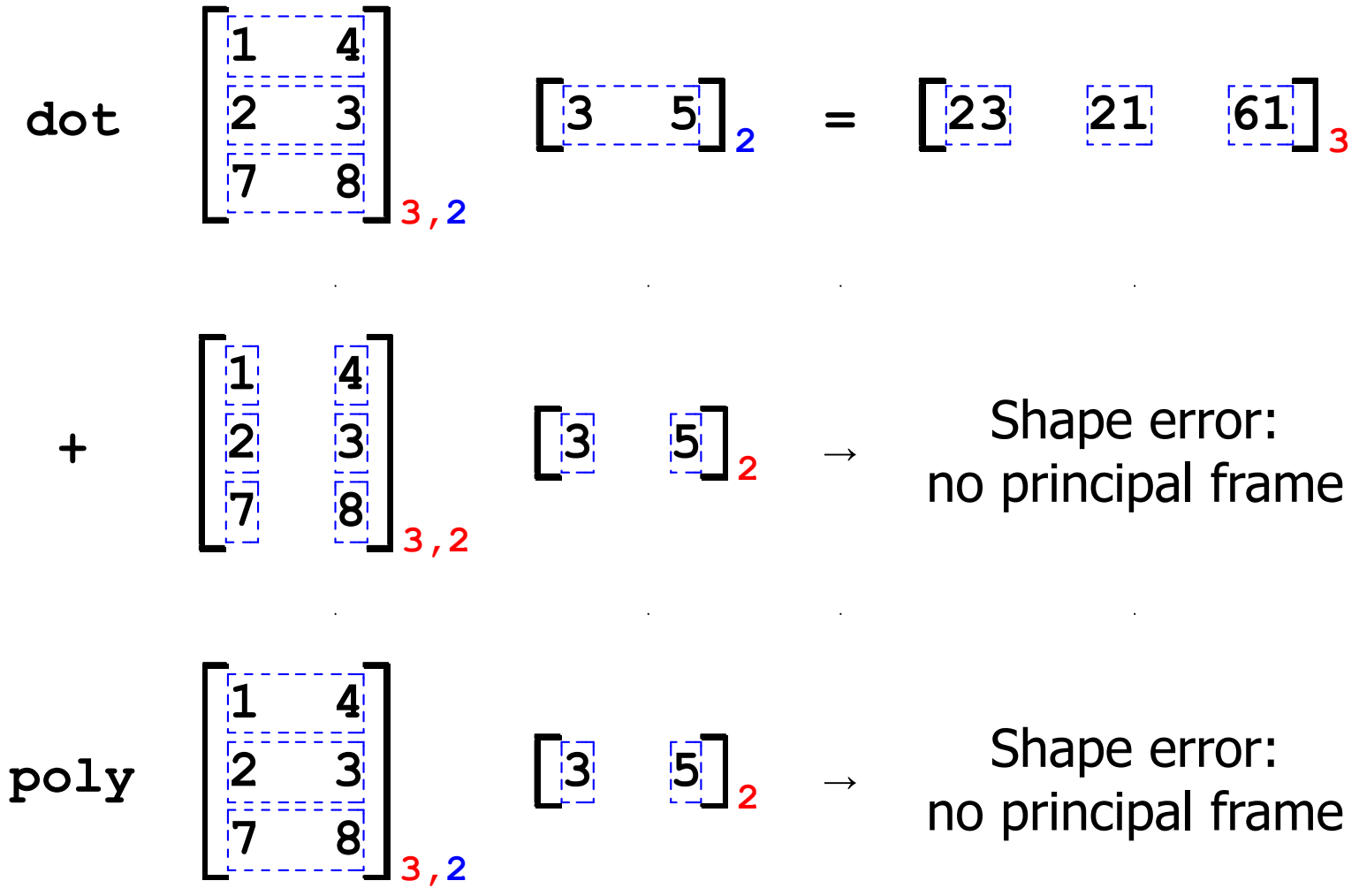
Trying different shapes



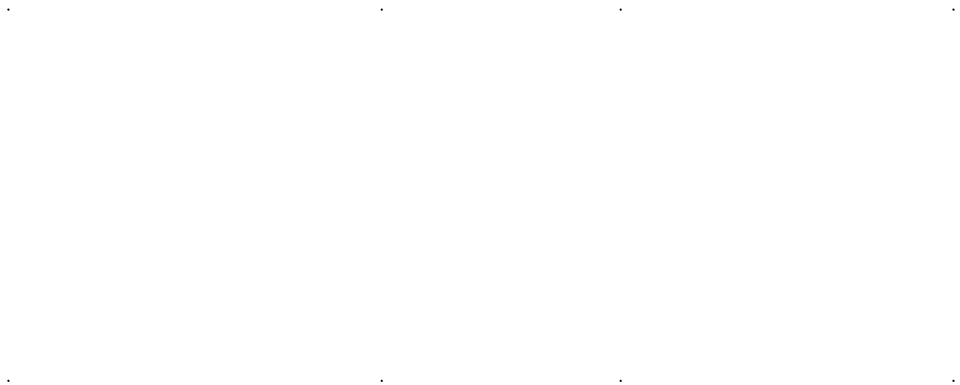
Trying different shapes



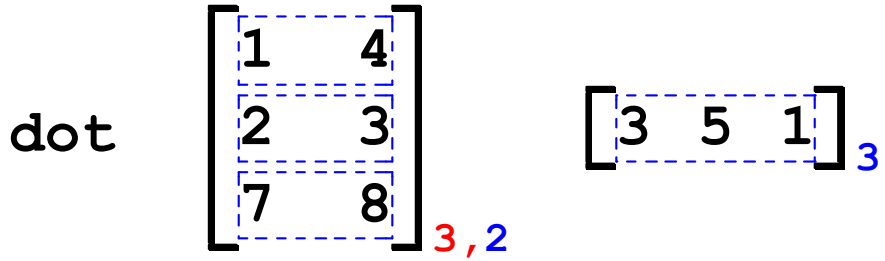
Trying different shapes



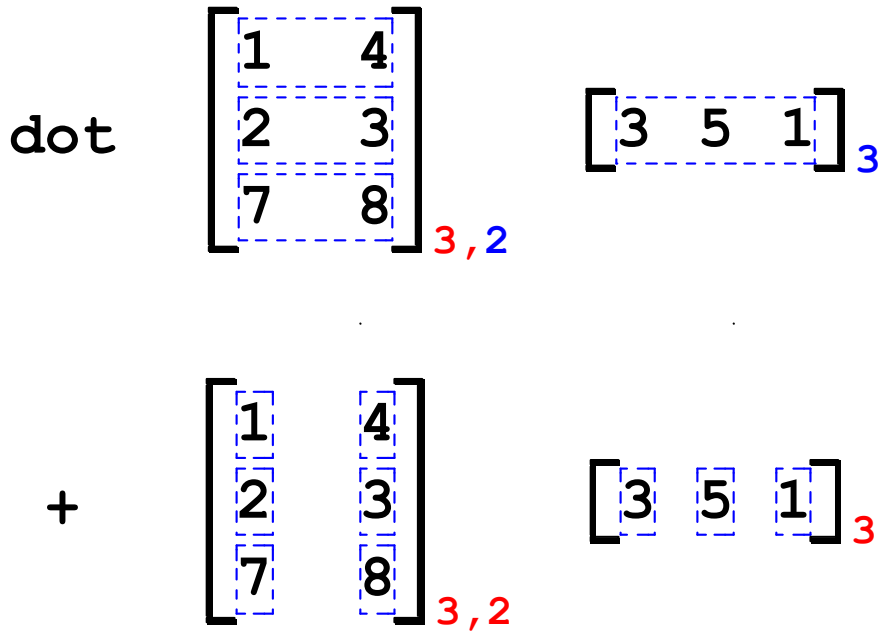
Trying different shapes



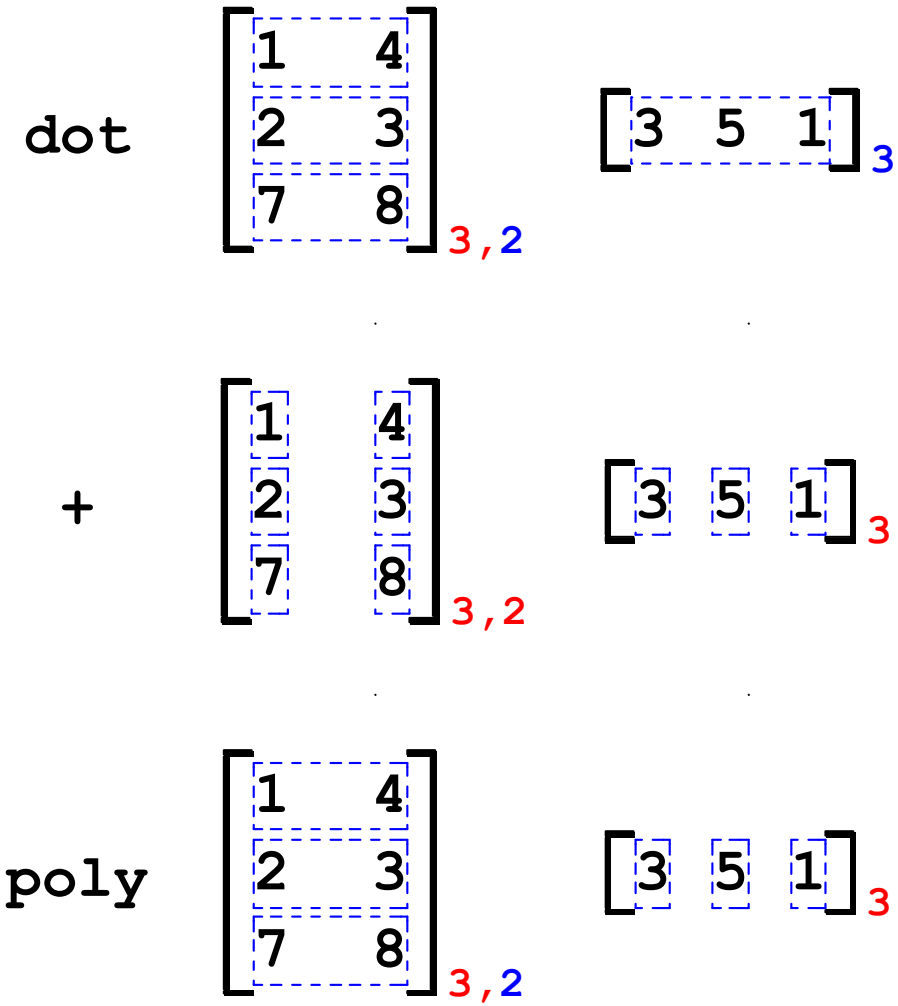
Trying different shapes



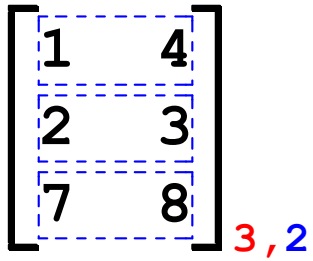
Trying different shapes



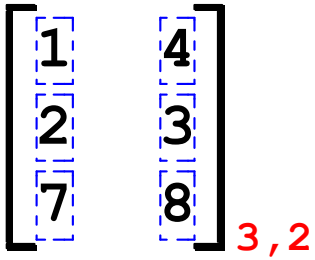
Trying different shapes

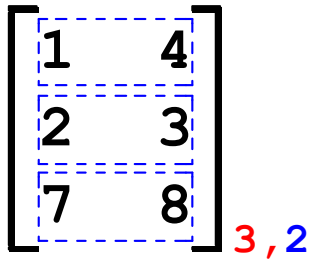


Trying different shapes

`dot` 

→ Domain error: `dot` requires equal length args

`+` 

`poly` 

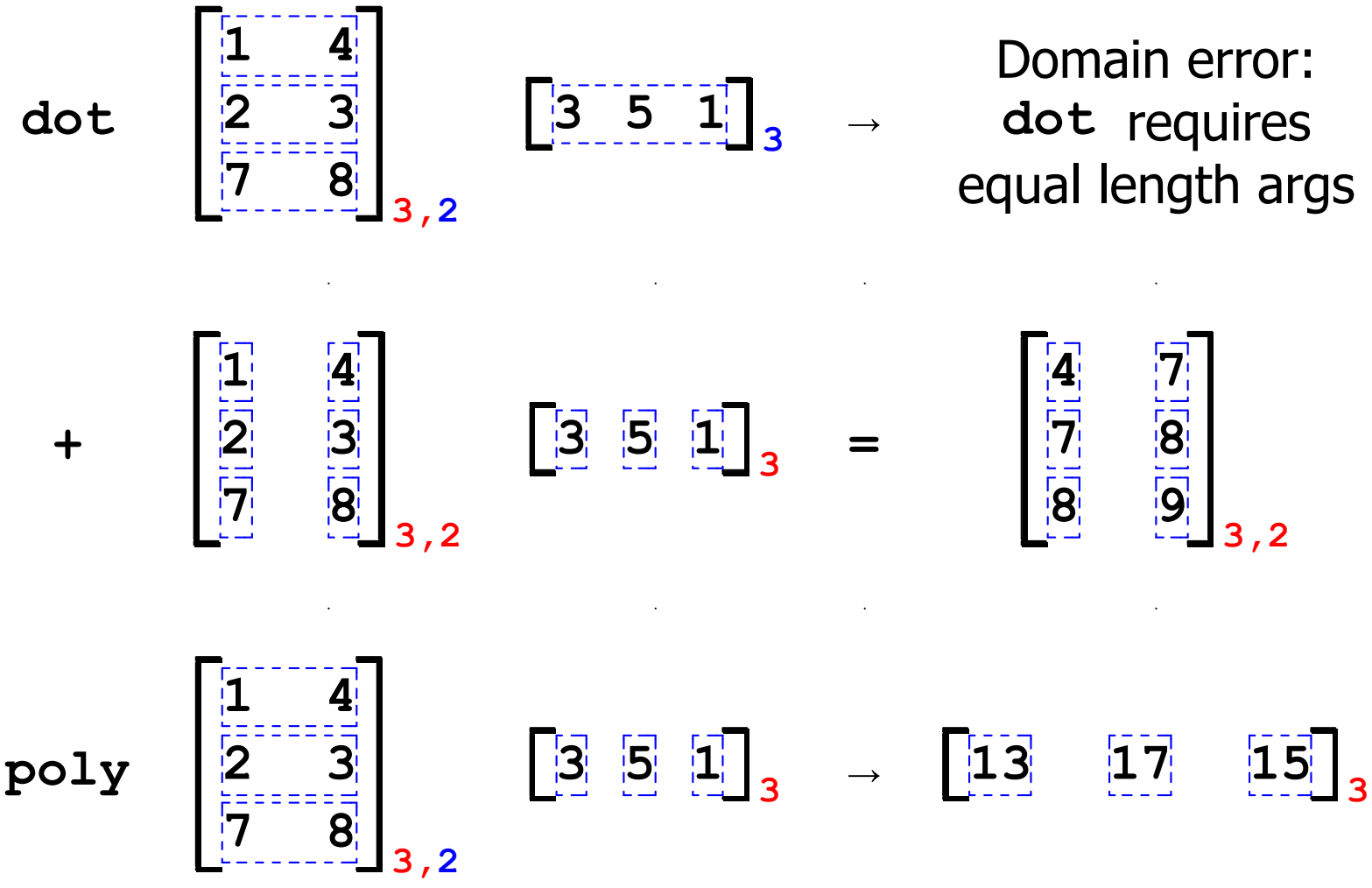
Trying different shapes

`dot` $\begin{bmatrix} 1 & 4 \\ 2 & 3 \\ 7 & 8 \end{bmatrix}_{3,2}$ $\begin{bmatrix} 3 & 5 & 1 \end{bmatrix}_{3}$ → Domain error: `dot` requires equal length args

`+` $\begin{bmatrix} 1 & 4 \\ 2 & 3 \\ 7 & 8 \end{bmatrix}_{3,2}$ $\begin{bmatrix} 3 & 5 & 1 \end{bmatrix}_{3}$ = $\begin{bmatrix} 4 & 7 \\ 7 & 8 \\ 8 & 9 \end{bmatrix}_{3,2}$

`poly` $\begin{bmatrix} 1 & 4 \\ 2 & 3 \\ 7 & 8 \end{bmatrix}_{3,2}$ $\begin{bmatrix} 3 & 5 & 1 \end{bmatrix}_{3}$

Trying different shapes



Typing Rank Polymorphic Code

$$\begin{bmatrix} 1 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 \end{bmatrix}_{3,7}$$

```
[[1 0 0 1 1 0 1]
 [0 1 0 1 0 1 1]
 [0 0 1 0 1 1 1]]
```

```
[[1 0 0 1 1 0 1]
 [0 1 0 1 0 1 1]
 [0 0 1 0 1 1 1]] : (Arr (Shp 3 7) Int)
```

```
[[1 0 0 1 1 0 1]
 [0 1 0 1 0 1 1]
 [0 0 1 0 1 1 1]] : (Arr (Shp 3 7) Int)
```

$$\left[\begin{array}{cc} 0 & 1 \\ 2 & 3 \\ \hline 4 & 5 \\ 6 & 7 \end{array}\right]_{2,2,2}$$

```
[[1 0 0 1 1 0 1]
 [0 1 0 1 0 1 1]
 [0 0 1 0 1 1 1]] : (Arr (Shp 3 7) Int)
```

```
[[[0 1]
  [2 3]]
 [[4 5]
  [6 7]]]
```

```
[[1 0 0 1 1 0 1]
 [0 1 0 1 0 1 1]
 [0 0 1 0 1 1 1]] : (Arr (Shp 3 7) Int)
```

```
[[[0 1]
  [2 3]]
 [[4 5]
  [6 7]]] : (Arr (Shp 2 2 2) Int)
```

```
[[1 0 0 1 1 0 1]
 [0 1 0 1 0 1 1]
 [0 0 1 0 1 1 1]] : (Arr (Shp 3 7) Int)
```

```
[[[0 1]
  [2 3]]
 [[4 5]
  [6 7]]] : (Arr (Shp 2 2 2) Int)
```

```
[0].
```

```
[[1 0 0 1 1 0 1]
 [0 1 0 1 0 1 1]
 [0 0 1 0 1 1 1]] : (Arr (Shp 3 7) Int)
```

```
[[[0 1]
  [2 3]]
 [[4 5]
  [6 7]]] : (Arr (Shp 2 2 2) Int)
```

```
0
```

```
[[1 0 0 1 1 0 1]
 [0 1 0 1 0 1 1]
 [0 0 1 0 1 1 1]] : (Arr (Shp 3 7) Int)
```

```
[[[0 1]
  [2 3]]
 [[4 5]
  [6 7]]] : (Arr (Shp 2 2 2) Int)
```

```
0 : (Arr (Shp) Int)
```

```
   +

 dot

poly
```

```
+ : (-> ((Arr (Shp) Int) (Arr (Shp) Int))
        (Arr (Shp) Int))
```

```
dot
```

```
poly
```

```
   + : (-> ((Arr (Shp) Int) (Arr (Shp) Int))
           (Arr (Shp) Int))

 dot : (Π ((l Dim))
         (-> ((Arr (Shp l) Int) (Arr (Shp l) Int))
             (Arr (Shp) Int)))

poly
```

```
   + : (-> ((Arr (Shp) Int) (Arr (Shp) Int))
           (Arr (Shp) Int))

 dot : (Π ((l Dim))
         (-> ((Arr (Shp l) Int) (Arr (Shp l) Int))
             (Arr (Shp) Int)))

poly : (Π ((l Dim))
         (-> ((Arr (Shp l) Int) (Arr (Shp) Int))
             (Arr (Shp) Int)))
```

```
      + : (-> ((Arr (Shp) Int) (Arr (Shp) Int))
              (Arr (Shp) Int))

    dot : (Π ((l Dim))
            (-> ((Arr (Shp l) Int) (Arr (Shp l) Int))
                (Arr (Shp) Int)))

   poly : (Π ((l Dim))
            (-> ((Arr (Shp l) Int) (Arr (Shp) Int))
                (Arr (Shp) Int)))

append
```

```
    + : (-> ((Arr (Shp) Int) (Arr (Shp) Int))
            (Arr (Shp) Int))

  dot : (Π ((l Dim))
          (-> ((Arr (Shp l) Int) (Arr (Shp l) Int))
              (Arr (Shp) Int)))

 poly : (Π ((l Dim))
          (-> ((Arr (Shp l) Int) (Arr (Shp) Int))
              (Arr (Shp) Int)))

append : (∀ ((T Atom))
          (Π ((l1 Dim) (l2 Dim) (c Shp))
            (-> ((Arr (++ (Shp l1) c) T)
                 (Arr (++ (Shp l2) c) T))
                (Arr (++ (Shp (+ l1 l2)) c) T))))
```

```
(+   [[1 4]
      [2 3]
      [7 8]] [3 5 1])
```

```
(+  [[1 4]
    [2 3]
    [7 8]] [3 5 1])

+ : (Arr (Shp)
         (-> ((Arr (Shp) Int)
              (Arr (Shp) Int))
             (Arr (Shp) Int)))
```

```
          (+  [[1 4]
               [2 3]
               [7 8]] [3 5 1])

        + : (Arr (Shp)
                 (-> ((Arr (Shp) Int)
                      (Arr (Shp) Int))
                     (Arr (Shp) Int)))

[[1 4]  : (Arr (Shp 3 2) Int)
 [2 3]
 [7 8]]
```

```
(+  [[1 4]
     [2 3]
     [7 8]] [3 5 1])

+ : (Arr (Shp)
         (-> ((Arr (Shp) Int)
              (Arr (Shp) Int))
             (Arr (Shp) Int)))

[[1 4]  : (Arr (Shp 3 2) Int)
 [2 3]
 [7 8]]

[3 5 1] : (Arr (Shp 3) Int)
```

```
(+  [[1 4]
     [2 3]
     [7 8]] [3 5 1])

+ : (Arr (Shp)
         (-> ((Arr (Shp) Int)
              (Arr (Shp) Int))
             (Arr (Shp) Int)))

[[1 4]  : (Arr (Shp 3 2) Int)
 [2 3]
 [7 8]]

[3 5 1] : (Arr (Shp 3) Int)
```

$$(\mathrm{Shp}) \sqsubseteq (\mathrm{Shp}\ 3) \sqsubseteq (\mathrm{Shp}\ 3\ 2)$$

```
(+  [[1 4]
     [2 3]
     [7 8]] [3 5 1])

+ : (Arr (Shp)
         (-> ((Arr (Shp) Int)
              (Arr (Shp) Int))
             (Arr (Shp) Int)))

[[1 4]  : (Arr (Shp 3 2) Int)
 [2 3]
 [7 8]]

[3 5 1] : (Arr (Shp 3) Int)
```

$$(\mathrm{Shp}) \sqsubseteq (\mathrm{Shp}\ 3) \sqsubseteq (\mathrm{Shp}\ 3\ 2)$$

```
(Arr (Shp 3 2) Int)
```

```
          (+  [[1 4]
               [2 3]
               [7 8]] [3 5 1])

[[+ +]  : (Arr (Shp 2 3)
 [+ +]         (-> ((Arr (Shp) Int)
 [+ +]]             (Arr (Shp) Int))
                   (Arr (Shp) Int)))

[[1 4]  : (Arr (Shp 3 2) Int)
 [2 3]
 [7 8]]

[[3 3]  : (Arr (Shp 3 2) Int)
 [5 5]
 [1 1]]
```

$$(\text{Shp}) \sqsubseteq (\text{Shp } 3) \sqsubseteq (\text{Shp } 3\ 2)$$

(Arr (Shp 3 2) Int)

```
(+   [[1 4]
      [2 3]
      [7 8]] [3 5])
```

```
      (+  [[1 4]
           [2 3]
           [7 8]] [3 5])

[[1 4]  : (Arr (Shp 3 2) Int)
 [2 3]
 [7 8]]

 [3 5] : (Arr (Shp 2) Int)
```

```
(+  [[1 4]
    [2 3]
    [7 8]] [3 5])
```

```
[[1 4]  : (Arr (Shp 3 2) Int)
 [2 3]
 [7 8]]

 [3 5] : (Arr (Shp 2) Int)
```

$$(\texttt{Shp}\ 2) \not\sqsubseteq (\texttt{Shp}\ 3\ 2)$$

```
(+  [[1 4]
    [2 3]
    [7 8]] [3 5])
```

```
[[1 4]  : (Arr (Shp 3 2) Int)
 [2 3]
 [7 8]]

 [3 5] : (Arr (Shp 2) Int)
```

$(\texttt{Shp 2}) \not\sqsubseteq (\texttt{Shp 3 2})$ $\quad$ $(\texttt{Shp 3 2}) \not\sqsubseteq (\texttt{Shp 2})$

```
(+  [[1 4]
    [2 3]
    [7 8]] [3 5])
```

```
[[1 4]  : (Arr (Shp 3 2) Int)
 [2 3]
 [7 8]]

 [3 5] : (Arr (Shp 2) Int)
```

$(Shp\ 2) \not\sqsubseteq (Shp\ 3\ 2)$ $(Shp\ 3\ 2) \not\sqsubseteq (Shp\ 2)$

Ill-typed

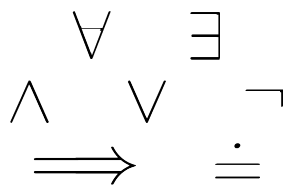
Constraints for Rank Polymorphism

First-order logic

First-order logic

$$\begin{array}{cccc} & \forall & \exists & \\ \wedge & & \vee & \neg \\ & \Longrightarrow & \doteq & \end{array}$$

First-order logic



Free monoid over $\mathbb{N}$

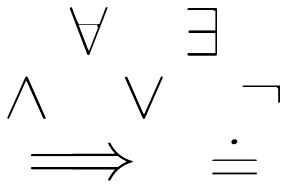
First-order logic

$$\forall \quad \exists$$
$$\wedge \quad \vee \quad \neg$$
$$\Longrightarrow \quad \doteq$$

Free monoid over $\mathbb{N}$

$$+\!+ \qquad \square$$

First-order logic



Free monoid over $\mathbb{N}$

$+\!+$ associativity

$\square$ identity

First-order logic

$\forall \quad \exists$

$\wedge \quad \vee \quad \neg$

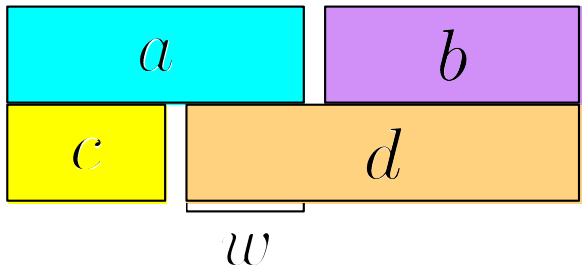
$\Longrightarrow \quad \doteq$

Free monoid over $\mathbb{N}$

$+\!\!+$ associativity

$\square$ identity

nothing equal unless required



First-order logic

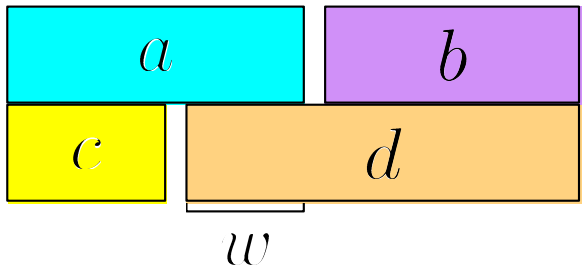
$$\forall \quad \exists$$
$$\wedge \quad \vee \quad \neg$$
$$\Longrightarrow \quad \doteq$$

Free monoid over $\mathbb{N}$

$$+\!+ \qquad \square \qquad +, 0, 1, \ldots$$

associativity identity

nothing equal unless required



$$prefix(x, y)$$

$$suffix(x, y)$$

$$prefix(x, y) \triangleq \exists z.x +\!\!+ z \doteq y$$

$$suffix(x, y)$$

$$prefix(x, y) \triangleq \exists z.x +\!\!+ z \doteq y$$

$$suffix(x, y) \triangleq \exists z.z +\!\!+ x \doteq y$$

$$prefix(x, y) \triangleq \exists z.x +\!\!+ z \doteq y$$

$$suffix(x, y) \triangleq \exists z.z +\!\!+ x \doteq y$$

Redundant but will be useful later

Type checking:

Validate index arguments supplied by programmer

Type checking:

Validate index arguments supplied by programmer

Cell types?

Type checking:

Validate index arguments supplied by programmer

Cell types?

$$a \doteq f +\!\!+ c$$

Type checking:

Validate index arguments supplied by programmer

Cell types?

$$a \doteq f +\!\!+ c$$

Principal frame?

Type checking:

Validate index arguments supplied by programmer

Cell types?

$$a \doteq f +\!\!+ c$$

Principal frame?

$$p \doteq f +\!\!+ e$$

Naïve solution

Naïve solution

- $\forall$ quantifiers for bound index vars

Naïve solution

- $\forall$ quantifiers for bound index vars
- $\exists$ quantifier for principal frame

Naïve solution

- $\forall$ quantifiers for bound index vars
- $\exists$ quantifier for principal frame
- $\exists$ quantifiers for prefix/suffix completions

Naïve solution

- $\forall$ quantifiers for bound index vars
- $\exists$ quantifier for principal frame
- $\exists$ quantifiers for prefix/suffix completions

$$\forall \overrightarrow{x} \exists \overrightarrow{f}, \overrightarrow{e}, p. \bigwedge_{i=1}^{n} (a_i \doteq f_i +\!\!+ c_i) \wedge \bigwedge_{i=1}^{n} (p \doteq f_i +\!\!+ e_i) \wedge \bigvee_{i=1} (p \doteq f_i)$$

Naïve solution

- $\forall$ quantifiers for bound index vars
- $\exists$ quantifier for principal frame
- $\exists$ quantifiers for prefix/suffix completions

$$\forall \overrightarrow{x} \exists \overrightarrow{f}, \overrightarrow{e}, p. \bigwedge_{i=1}^{n} (a_i \doteq f_i +\!\!+ c_i) \wedge \bigwedge_{i=1}^{n} (p \doteq f_i +\!\!+ e_i) \wedge \bigvee_{i=1}^{n} (p \doteq f_i)$$

Write p in terms of $\overrightarrow{x}$

Naïve solution

- $\forall$ quantifiers for bound index vars
- $\exists$ quantifier for principal frame
- $\exists$ quantifiers for prefix/suffix completions

$$\forall \overrightarrow{x} \exists \overrightarrow{f}, \overrightarrow{e}, p.\ \bigwedge_{i=1}^{n} (a_i \doteq f_i +\!\!+ c_i)$$
$$\wedge \bigwedge_{i=1}^{n} (p \doteq f_i +\!\!+ e_i)$$
$$\wedge \bigvee_{i=1}^{n} (p \doteq f_i)$$

Write p in terms of $\overrightarrow{x}$ Decidability issues

Better solution

Better solution

Canonicalize symbolic shapes

Better solution

Canonicalize symbolic shapes

$$[2, n+2] +\!\!+ [6] +\!\!+ d +\!\!+ [2+n]$$

Better solution

Canonicalize symbolic shapes

$[2, n + 2] +\!\!+ [6] +\!\!+ d +\!\!+ [2 + n]$ $[2] +\!\!+ [1 + n + 1, 6] +\!\!+ d +\!\!+ [2 + n]$

Better solution

Canonicalize symbolic shapes

$[2, n+2] +\!\!+ [6] +\!\!+ d +\!\!+ [2+n]$ $\qquad$ $[2] +\!\!+ [1+n+1, 6] +\!\!+ d +\!\!+ [2+n]$

$$[2] +\!\!+ [n+2] +\!\!+ [6] +\!\!+ d +\!\!+ [n+2]$$

In universal fragment, easy equality/prefix/suffix — including witnesses

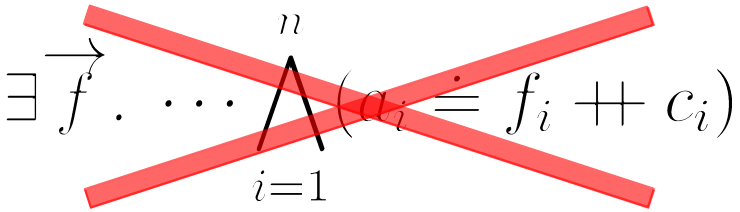
In universal fragment, easy equality/prefix/suffix — including witnesses

$$\exists \overrightarrow{f} . \ \cdots \bigwedge_{i=1}^{n} (a_i \doteq f_i +\!\!+ c_i)$$

In universal fragment, easy equality/prefix/suffix — including witnesses

$$\exists \overrightarrow{f} . \cdots \bigwedge_{i=1}^{n} (a_i \doteq f_i +\!\!+ c_i)$$

In universal fragment, easy equality/prefix/suffix — including witnesses



Fold argument frame shapes (suffix witnesses) together

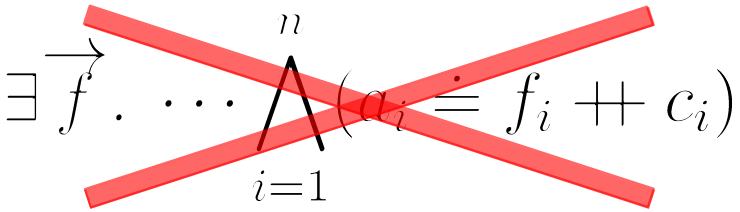
In universal fragment, easy equality/prefix/suffix — including witnesses

$$\exists \overrightarrow{f} . \cdots \bigwedge_{i=1}^{n} (a_i \doteq f_i +\!\!+ c_i)$$

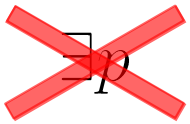
Fold argument frame shapes (suffix witnesses) together

$$\exists p$$

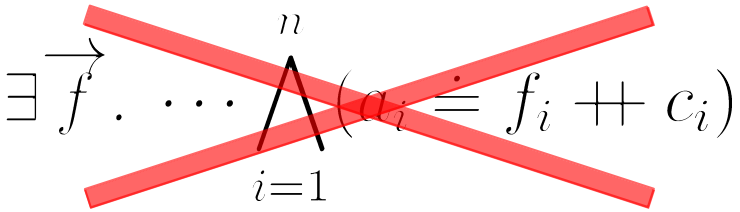
In universal fragment, easy equality/prefix/suffix — including witnesses



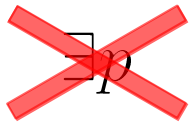
Fold argument frame shapes (suffix witnesses) together



In universal fragment, easy equality/prefix/suffix — including witnesses



Fold argument frame shapes (suffix witnesses) together



Type checking works entirely within universal fragment

Type inference:

Identify index arguments elided by programmer

Type inference:

Identify index arguments elided by programmer

Function type

```
(Pi (e ...)
  (-> ((Arr ι σ)
       ...)
      τ))
```

Type inference:

Identify index arguments elided by programmer

Function type

```
(Pi (e ...)
  (-> ((Arr ι σ)
       ...)
      τ))
```

Argument types

```
(Arr κ σ) ...
```

Type inference:

Identify index arguments elided by programmer

Function type

```
(Pi (e ...)
  (-> ((Arr ι σ)
       ...)
      τ))
```

Argument types

```
(Arr κ σ) ...
```

- $\forall$ quantifiers for bound index vars

Type inference:

Identify index arguments elided by programmer

Function type

```
(Pi (e ...)
  (-> ((Arr ι σ)
       ...)
      τ))
```

Argument types

```
(Arr κ σ) ...
```

- ∀ quantifiers for bound index vars
- ∃ quantifiers for frames

Type inference:

Identify index arguments elided by programmer

Function type

```
(Pi (e ...)
  (-> ((Arr ι σ)
       ...)
      τ))
```

Argument types

```
(Arr κ σ) ...
```

- ∀ quantifiers for bound index vars
- ∃ quantifiers for frames
- ∃ quantifiers for index arguments

Type inference:

Identify index arguments elided by programmer

Function type

```
(Pi (e ...)
  (-> ((Arr ι σ)
       ...)
      τ))
```

Argument types

```
(Arr κ σ) ...
```

- ∀ quantifiers for bound index vars
- ∃ quantifiers for frames (eliminated before)
- ∃ quantifiers for index arguments

Type inference:

Identify index arguments elided by programmer

Function type

```
(Pi (e ...)
  (-> ((Arr ι σ)
       ...)
      τ))
```

Argument types

```
(Arr κ σ) ...
```

- $\forall$ quantifiers for bound index vars
- $\exists$ quantifiers for frames (eliminated before)
- $\exists$ quantifiers for index arguments (solving for these)

Function type

```
(Pi (e ...)
  (-> ((Arr ι σ)
       ...)
      τ))
```

Argument types

```
(Arr κ σ) ...
```

Function type

```
(Pi (e ...)
  (-> ((Arr ι σ)
       ...)
      τ))
```

Argument types

```
(Arr κ σ) ...
```

$$\forall \overrightarrow{x} . \exists \overrightarrow{f}, \overrightarrow{e} . \bigwedge_{i=1}^{n} (f_i +\!\!+ \iota_i \doteq \kappa_i)$$

Function type

```
(Pi (e ...)
  (-> ((Arr ι σ)
       ...)
      τ))
```

Argument types

```
(Arr κ σ) ...
```

$$\forall \overrightarrow{x} . \exists \overrightarrow{f}, \overrightarrow{e} . \bigwedge_{i=1}^{n} (f_i +\!\!+ \iota_i \doteq \kappa_i)$$

$$\mathcal{I} \quad = \quad e_i \mapsto T_i(\overrightarrow{x})$$

Function type

```
(Pi (e ...)
  (-> ((Arr ι σ)
       ...)
      τ))
```

Argument types

```
(Arr κ σ) ...
```

$$\forall \overrightarrow{x} . \exists \overrightarrow{f}, \overrightarrow{e} . \bigwedge_{i=1}^{n} (f_i +\!\!+ \iota_i \doteq \kappa_i)$$

$$\mathcal{I} \quad = \quad e_i \mapsto T_i(\overrightarrow{x}) \qquad\qquad \Sigma = +\!\!+, \square, +$$

Function type

```
(Pi (e ...)
  (-> ((Arr ι σ)
       ...)
      τ))
```

Argument types

```
(Arr κ σ) ...
```

$$\forall \overrightarrow{x} . \exists \overrightarrow{f}, \overrightarrow{e} . \bigwedge_{i=1}^{n} (f_i +\!\!+ \iota_i \doteq \kappa_i)$$

$$\mathcal{I} \quad = \quad e_i \mapsto T_i(\overrightarrow{x}) \qquad\qquad \Sigma = +\!\!+, \square, +$$

Pretend $\overrightarrow{x}$ are additional generators

Why can we treat $\overrightarrow{l}$ like additional generators?

Why can we treat $\overrightarrow{\ell}$ like additional generators?

Algorithm for existential fragment (Makanin, 1977)

$$c +\!\!+ [5] +\!\!+ d +\!\!+ [5] +\!\!+ d \doteq d +\!\!+ [5] +\!\!+ [2] +\!\!+ d +\!\!+ e$$

Why can we treat $\overrightarrow{\ldots}$ like additional generators?

Algorithm for existential fragment (Makanin, 1977)

$$c +\!\!+ [5] +\!\!+ d +\!\!+ [5] +\!\!+ d \doteq d +\!\!+ [5] +\!\!+ [2] +\!\!+ d +\!\!+ e$$

"How might subsequences align with each other?"

Why can we treat $\overrightarrow{\ldots}$ like additional generators?

Algorithm for existential fragment (Makanin, 1977)

$$c +\!\!+ [5] +\!\!+ d +\!\!+ [5] +\!\!+ d \doteq d +\!\!+ [5] +\!\!+ [2] +\!\!+ d +\!\!+ e$$

"How might subsequences align with each other?"

Equations about smaller sequences

Why can we treat $\overrightarrow{\ell}$ like additional generators?

Algorithm for existential fragment (Makanin, 1977)

$$c +\!\!+ [5] +\!\!+ d +\!\!+ [5] +\!\!+ d \doteq d +\!\!+ [5] +\!\!+ [2] +\!\!+ d +\!\!+ e$$

"How might subsequences align with each other?"

Equations about smaller sequences

No boundaries *inside* a generator

Why can we treat $\overrightarrow{\ell}$ like additional generators?

Algorithm for existential fragment (Makanin, 1977)

$$c +\!\!+ [5] +\!\!+ d +\!\!+ [5] +\!\!+ d \doteq d +\!\!+ [5] +\!\!+ [2] +\!\!+ d +\!\!+ e$$

"How might subsequences align with each other?"

Equations about smaller sequences

No boundaries *inside* a generator ... or a universal variable

Why can we treat $\overrightarrow{\ldots}$ like additional generators?

Algorithm for existential fragment (Makanin, 1977)

$$c +\!\!+ [5] +\!\!+ d +\!\!+ [5] +\!\!+ d \doteq d +\!\!+ [5] +\!\!+ [2] +\!\!+ d +\!\!+ e$$

"How might subsequences align with each other?"

Equations about smaller sequences

No boundaries *inside* a generator ... or a universal variable

Caveat: only works in conjunctive fragment

Multiple solutions?

Multiple solutions?

```
(append
 [[1 2]
  [3 4]]
 [[5 6]
  [7 8]])
```

Multiple solutions?

```
(append
 [[1 2]
  [3 4]]
 [[5 6]
  [7 8]])
```

```
(Pi ((l1 Dim)
     (l2 Dim)
     (c Shape))
  (Arr (Shp)
       (-> ((Arr (++ (Shp l1) c) Int)
            (Arr (++ (Shp l2) c) Int))
           (Arr (++ (Shp (+ l1 l2)) c) Int))))
```

Multiple solutions?

```
(append
 [[1 2]
  [3 4]]
 [[5 6]
  [7 8]])
```

```
(Pi ((l1 Dim)
     (l2 Dim)
     (c Shape))
  (Arr (Shp)
       (-> ((Arr (++ (Shp l1) c) Int)
            (Arr (++ (Shp l2) c) Int))
           (Arr (++ (Shp (+ l1 l2)) c) Int))))
```

```
c = (Shp 2)
c = (Shp)
```

Multiple solutions?

```
(append
 [[1 2]
  [3 4]]
 [[5 6]
  [7 8]])
```

```
(Pi ((l1 Dim)
     (l2 Dim)
     (c Shape))
  (Arr (Shp)
       (-> ((Arr (++ (Shp l1) c) Int)
            (Arr (++ (Shp l2) c) Int))
           (Arr (++ (Shp (+ l1 l2)) c) Int))))
```

```
c = (Shp 2)
```

Major axis append

```
c = (Shp)
```

Multiple solutions?

```
(append
 [[1 2]
  [3 4]]
 [[5 6]
  [7 8]])
```

```
(Pi ((l1 Dim)
     (l2 Dim)
     (c Shape))
  (Arr (Shp)
       (-> ((Arr (++ (Shp l1) c) Int)
            (Arr (++ (Shp l2) c) Int))
           (Arr (++ (Shp (+ l1 l2)) c) Int))))
```

`c = (Shp 2)` Major axis append

`c = (Shp)` Lifted row-wise append

Multiple solutions?

```
(append
 [[1 2]
  [3 4]]
 [[5 6]
  [7 8]])
```

```
(Pi ((l1 Dim)
     (l2 Dim)
     (c Shape))
  (Arr (Shp)
       (-> ((Arr (++ (Shp l1) c) Int)
            (Arr (++ (Shp l2) c) Int))
           (Arr (++ (Shp (+ l1 l2)) c) Int))))
```

`c = (Shp 2)` Major axis append

`c = (Shp)` Lifted row-wise append

Established convention: Operate along major axis

Multiple solutions?

```
(append
 [[1 2]
  [3 4]]
 [[5 6]
  [7 8]])
```

```
(Pi ((l1 Dim)
     (l2 Dim)
     (c Shape))
  (Arr (Shp)
       (-> ((Arr (++ (Shp l1) c) Int)
            (Arr (++ (Shp l2) c) Int))
           (Arr (++ (Shp (+ l1 l2)) c) Int))))
```

`c = (Shp 2)` Major axis append

`c = (Shp)` Lifted row-wise append

Established convention: Operate along major axis

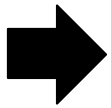
Require scalar frame shape

Remaining work

Remaining work

Remaining work

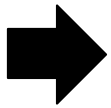
Generating

$\forall^*\exists^*(\wedge)$  $\exists*$

Remaining work

Generating

$\forall^*\exists^*(\wedge)$



Solving

$\exists*$

Remaining work

Solving
$\exists *$

Largely done (not by me — Makanin, 1977; Karhumäki, et al, 2000)

ILP modulo theories (Manolios & Papavasileiou, 2013)

Remaining work

Generating

$\forall^*\exists^*(\wedge)$

In progress

Rank Polymorphism Viewed as a Constraint Problem

Justin Slepak
jrslepak@ccs.neu.edu

Panagiotis Manolios
pete@ccs.neu.edu

Olin Shivers
shivers@ccs.neu.edu

Northeastern University

Questions?